A SOFTWARE DEVELOPMENT APPROACH FOR PARALLEL PROCESSING SYSTEMS INTEGRATING OBJECT ORIENTED AND FUNCTIONAL PARADIGMS

BY

DOO HWAN BAE

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1992

To my parents, wife and daughter

AGKNOWLEDGMENTS

I would like to express my gratitude to my advisor Professor Stephen S Yau for his advice and support in my Ph D program at the University of Florida and Northwestern University Throughout my graduate study, he made contributions in various ways advice, insight, alternative solutions and encouragement I would like to thank my committee members, Professor Yann Hang Lee, Professor Paul W Chun, Professor Paul A Fishwick and Professor Tim Davis Professor Lee helped me continue my work by not only giving technical advice but also providing spiritual support Professor Chun also provided me with friendly advice and encouragement. The discussions with him regarding parallel processing have always been productive for my study Advice from Dr Xiaoping Jia was invaluable to my research I have enjoyed working with all of them Special thanks are due to my colleagues in our software engineering research group, G Oh, M Chidambaram, K Yeom, W Sung, V Satish, R. Yang and G Pour. Informal discussions with them also contributed to this work I also would like to acknowledge the partial support from Rome Laboratory, U S Air Force Systems Command for this research

I wish to thank my parents for their endless support for the long years of my school life. I also wish to thank my wife, Eun Young, for her continuous encouragement and patience. Without her help, I would not be able to complete my Ph.D degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF NOTATIONS

Symbol	Description
A	a labeled transition system
$A_T(\mathcal{P})$	an interleaving semantics of the program P
c(x,y)	a communication time between node x and y
D	a set of final places in the Petri net
e(x)	an execution time required to complete a node x
E	a set of arcs in the method level IPR.
E(c,n)	a execution time to complete a given task c with n processors
E(s_i)	one-pass completion time for segment s_i in a pipe-lined process.
F	a set of arcs representing flow relations in the Petri-net.
G	a directed graph representing the method level IPR.
$GAIN(a, \ldots, b)$	a gain obtained by clustering nodes a, , b
G_g	a gain graph
G_p	a task precedence graph
I	a set of initial places in the Petri net
N	a general net
P	a set of places in the Petri-net
$\bullet p$	a preset of a place p.
$p\bullet$	a postset of a place p
$\mathcal{PN}$	a Petri net used for the object level IPR.
$\mathcal{R}$	a pseudo-function to modify the object state
$\bullet t$	a preset of a transition t
$t\bullet$	a postset of a transition t

T	a set of transitions in the Petri net.
T_g	a gain tree
T_p	a task precedence tree.
V	a set of nodes in the method level IPR.

Abstract of Dissertation Presented to the Graduate School of
the University of Florida in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy

A SOFTWARE DEVELOPMENT APPROACH FOR PARALLEL PROCESSING SYSTEMS INTEGRATING OBJECT ORIENTED AND FUNCTIONAL PARADIGMS

by
Doo-Hwan Bae

May 1992

Chairman Stephen S. Yau
Major Department Computer and Information Sciences

Although there are many parallel computers available, the effective use of their computing power is difficult to achieve due to the lack of proper software development methods and techniques for parallel processing systems. In this dissertation, we present a computation model, called PROOF (PaRallel Object Oriented Functional) as a basis for expressing parallelism, and an intermediate form, called IPR (Intermediate Program Representation) as a basis for exploiting parallelism in software development for parallel processing systems. We also present grain size determination techniques on different types of parallelism. We develop the computation model PROOF in which the object-oriented paradigm is integrated with the functional paradigm without sacrificing the advantages of either. The major features of our computation model include expressing the various granularity levels of parallelism, integrating the referential transparency and history sensitivity, and supporting inheritance and synchronization without interference. The computation model PROOF serves as a basis for developing our approach to the software development for parallel processing systems. We develop the IPR, which is used for representing all the parallelism in the software and performing partitioning analysis. We develop

transformation rules from a PROOF code to the IPR and from the IPR to a target code and show that the transformation rules preserve the correctness of the original PROOF programs. For efficient exploitation of parallelism, we present the grain size determination algorithms on various patterns of parallelism and compare them with the existing approaches. Software development based on PROOF has the following distinct advantages. First, parallel aspects of the software can be treated as a prime issue, using object oriented concepts. Second, our approach can reduce the communication and synchronization errors by automatically embedding proper codes during the transformation stage. Third, the use of the IPR improves portability of the software by separating architecture-dependent issues from the semantics of the software.

CHAPTER I
INTRODUCTION

With the advances in computer technology, the speed of computers has significantly increased over the last several decades. However, even such vastly increased speed is not sufficient to fulfill our desire to solve complex problems, involving artificial intelligence, robotics, high energy physics, molecular physics and space sciences. Parallel processing appears to be a promising solution to satisfy such problems. Unfortunately, software development methods and techniques for parallel processing systems are far from being mature enough to effectively utilize their potential. The lack of such methods and techniques is a major obstacle to the widespread use of parallel processors in a variety of application areas. The goal of this research is to develop a software development approach for parallel processing systems and study the issues involved in such development.

Software development is more complex for parallel processors than for sequential computers in at least two aspects: program correctness and efficiency. One of the requirements for the correctness of the parallel programs is the independence of the execution results on the number and speed of processors running the program. This requirement can be met only if the parallel tasks are properly synchronized. The focus on efficiency in parallel program design increases the complexity of software development. The efficiency of parallel implementation is usually measured by dividing the speed-up of the parallel program relative to its sequential version with

the number of processors involved in that parallel program execution. Thus, in order to achieve a reasonable performance in terms of efficiency, proper techniques for software development for parallel processing systems need to be developed.

1.1 Management of Parallelism

Parallel processing of a program is based on the following two assumptions:

- the availability of the parallel computers that can cooperate to solve a given problem,
- the existence of program development techniques such as design techniques for parallel programs and efficient implementation strategies.

In fact, as cost effective parallel computers become more available on the market, greater computing power can be easily obtained. Thus, the first assumption seems to be met. However, current software development techniques for parallel processing systems are not adequate to specifically address the issues involved in programming parallel processors, most importantly parallelism.

The essence of parallel processing is to manage parallelism. Parallelism refers to the execution of some portions of a program in a simultaneous manner. The management of the parallelism involves two issues:

- Expressing parallelism: for parallel processing of a program, the parallelism in that program needs to be represented using a programming language, and
- Exploiting parallelism: the parallelism needs to be efficiently exploited on the underlying parallel computers.

Parallelism can be specified either implicitly or explicitly. Explicit parallelism is expressed using explicit parallel constructs by the programmers. Thus, the detection of the explicit parallelism is easy. However, the programmers have to ensure

the correct use of such constructs in terms of communication and synchronization of parallel programs. In contrast, implicit parallelism implies that programmers are not concerned about the parallelism. In this case, the underlying computation models inherently support the parallelism. Thus, programmers do not use any explicit parallel constructs. On the other hand, either a compiler or run time system requires to detect parallelism in the programs. Once the parallelism is explicitly expressed and detected, it needs to be exploited on parallel computers. The exploitation techniques vary depending on the kind of parallelism and the parallel computers involved. Expressing and exploiting parallelism in the software is most important part of software development for parallel processing systems and is the subject of this study.

1.2 Parallel Computers

In the section, we briefly summarize the existing parallel computers. These can be divided into two categories: special purpose and general-purpose. The special-purpose parallel computers are targeted to solve specific problems. The vector processors in super computers, such as Fujitsu VP 200 and Hitachi S-810[2], have been developed to exploit parallelism mainly within loops. In these computers, the vector processors are controlled by a main control unit. Processor arrays consist of a number of identical processors under the control of a common control unit. Each processor has access to its own data, and thus the same operation can be performed simultaneously on many data items. In Flynn's classification [29], this kind of computer belongs to SIMD (Single Instruction Stream, Multiple Data Stream). The parallel computers belonging to this category include Connection Machine [39], DAP (Distributed Array Processors) [41] and IBM GF11 [11]. Systolic arrays [53] consist of

[2]These computers can be classified as general purpose computers. However, because their support to parallel processing is limited only to vector processing, we classify them as special purpose computers.

two or more dimensions of array processors in which each processor has its own control to perform specific task The data stream passes through the array processors to produce a final result

In comparison to those special purpose parallel computers, the general purpose computers belong to MIMD (Multiple Instruction Stream, Multiple Data Stream) according to Flynn's classification The parallel computers in this category are a set of asynchronous parallel processors These computers can be further divided into shared memory parallel computers and distributed memory computers. Because the focus of this research is to develop a parallel programming approach for general-purpose parallel processors, we will further examine each of the two kinds of parallel computers

An MIMD shared-memory parallel computer consists of a number of processors all having access to a single shared memory The processors communicate by *read* and *write* operations, and are connected to the shared memory via one or more shared buses or interconnection networks As the number of processors in the system increases, the communication medium becomes a bottleneck in terms of performance as well as cost Thus, a linear speed-up with an increase in the number of processors is not achievable or is limited to a certain number of processors In the MIMD shared-memory parallel computers, the major software design problems include data access synchronization and load balancing. The shared memory parallel computers include Cray X-MP and Y-MP series [45], Alliant FX/8 [76], Encore Multimax [45], IBM RP3 [66] and Sequent Balance [77]

To model the computation on the shared memory parallel computers, a theoretical model of parallel computation, called PRAM (Parallel Random Access Memory) [30] can be used In the PRAM model, each processor is a RAM (Random Access Machine) Processors communicate by reading from and writing to a global memory

The instructions in each processor are executed in a synchronous manner as if there were a global clock. PRAM models can be further divided according to the resolution schemes for the simultaneous read and write access. Exclusive Read Exclusive Write (EREW), Concurrent Read Exclusive Write (CREW), Exclusive Read Concurrent Write (ERCW) and Concurrent Read Concurrent Write (CRCW). Although the underlying assumptions in such PRAM models, such as constant communication time and synchronous execution of instruction, are not realistic, they have been used as the multiprocessor representations in the evaluation of parallel algorithms and task scheduling problems.

An MIMD distributed parallel computer consists of a set of processors each having a one shared local memory. Processors communicate by message passing via communication channels. In the MIMD distributed memory parallel computers, the synchronization is implicit through communication. The popular interconnection networks include *hypercube*, *ring*, *tree* and *mesh*. Unlike shared memory parallel computers, distributed memory parallel computers are not affected by the memory contention problem and are more easily expandable. One of the problems encountered by the distributed memory parallel computers is a message passing latency due to the possible transferring of data via intermediate processors. The major software design problems include data placement, communication overhead and scheduling. The shared memory parallel computers can be seen as a special form of distributed memory parallel computers in which all the processors are fully connected. Our software development approach is targeted for the MIMD distributed memory parallel computers. Any approach for the MIMD distributed memory parallel computers can be adopted to MIMD shared memory parallel computers with minor modifications. The MIMD distributed memory parallel computers include hypercube [35], NCUBE [64], BBN [16] and Inmos Transputer network [56].

1.3 Dissertation Outline

In this dissertation a software development approach for parallel processing systems is presented. The approach is based on an integrated object-oriented and functional paradigm. We develop a computation model PROOF (PaRallel Object-Oriented Functional) based on object-oriented and functional paradigms [84]. This PROOF computation model permits the expression of explicit and implicit parallelism using objects and functions in various levels of granularity. This computation model serves as a platform in developing our approach to programming parallel processors. It provides a framework for programmers to express parallelism in the application. Once parallelism is expressed using PROOF, it can be exploited when the PROOF program is transformed to a target code. The transformation is done in two steps. The first step transforms the PROOF program to the IPR which is independent of both a target language and a target machine. We do analysis on the IPR to determine the proper grain sizes. The second step transforms the IPR to a target code, using the information generated by the grain size analysis. The dissertation is organized as follows.

Chapter 2 surveys computation models and software development approaches for parallel processing systems[2]. Three parallel programming approaches–parallelizing compiler approach, programming language constructs approach and parallel programming language approach–are discussed. In particular, the existing parallel programming language approaches have been extensively discussed. Chapter 3 presents the PROOF computation model. In this chapter, the characteristics of the PROOF are presented along with examples. Major characteristics includes incorporation of object-oriented concepts to functional paradigm, allowing various granularity levels

[2]We have shortened the phrase 'software development for parallel processing systems' to 'programming parallel processors'.

of parallelism, integration of referential transparency with history sensitivity and integration of parallelism with inheritance. Chapter 4 presents the IPR and the rules for two-phase transformation of the PROOF/L[3] programs. A PROOF program is first transformed to the IPR code, and then the IPR code is transformed to a target code. We show that such a transformation preserves the correctness of the original program. Chapter 5 presents an allocation strategy for exploiting parallelism in programs based on the computation model PROOF. We develop grain size determination algorithms on various types of parallelism. In the case of pipe-lined parallelism, we show that our algorithm can find optimal solutions. In the cases of tree parallelism and graph parallelism, we introduce the concept of gain to analyze grain sizes. We also show that our approach performs better than the existing approaches. Chapter 6 summarizes the dissertation and discuss the directions of future research.

[3]The PROOF/L is a prototype programming language based on the computation model PROOF. The syntax of the PROOF/L is given in Appendix A1.

CHAPTER 2
COMPUTATION MODELS AND PARALLEL PROGRAMMING

In this chapter, we have surveyed the existing computation model and their supports to parallel programming, and the existing approaches to programming parallel processors. This chapter is organized as follows. In Section 2.1, the existing computation models and its extension for parallel processing are surveyed. In Section 2.2, the existing parallel programming approaches are compared, and the parallel language approach is identified as the most promising. In Section 2.3, the existing parallel programming approaches are surveyed. In Section 2.4, the contributions of this dissertation are outlined.

2.1 Existing Computation Models

As the various cost-effective MIMD parallel computers begin to appear on the market, more general approaches to programming such computers are required so that the programmers can write programs without concern about the details of the parallel computers. Because programming languages have been used as communication vehicles between the programmers and the computers since the appearance of the first computer, the programming languages in writing parallel programs play an important role in the parallel programming systems. More importantly, the underlying computation model on which the semantics of the programming language is based has significant impart on the parallel programming systems. There are many programming languages used in writing programs for these parallel processing systems. Since each of these programming languages has different features, their comparison

is not an easy task. Rather than examining each of these programming languages in terms of the specific features, we classify them into five different categories – *imperative, logic, functional, object oriented* and *knowledge-based* – according to their underlying computation model

2.1.1 Imperative Computation Model

As Backus [6] described their programming style as one-word-at-a-time style, the programming languages in this category have been significantly influenced by the von Neumann architecture. In imperative programming, computation is achieved by the repeated and step by step computation of low level values and the assignment of these values in memory locations. Programmers explicitly specify the order of instruction execution using control structures. This is also called as the von Neumann model of programming.

There have been ways to use the programming languages in this model for parallel processing. In one, the programmers write sequential programs and let parallelizing compilers detect parallelism in the programs. This has been widely used primarily because the programmers need not make additional effort in making their programs to execute in parallel. However, these compilers can only detect localized parallelism within loops.

Another way is to extend existing imperative languages with parallel language constructs. Since the parallel constructs are added to inherently sequential programming languages, the programmers are heavily burdened with explicitly specifying parallelism, communication and synchronization among parallel units. Fortran [3, 20], C [5, 28], Ada [22], and Occam [46] belong to this category.

2.1.2 Logic Computation Model

In logic programming, the basic element is the clause A logic program consists of a set of clauses goal clause, assertion clause and conditional clauses declaring the goal, fact and properties that describe the problem for which the solution is sought. The implementation system(interpreter), then, uses a unification process to show if the goal clause is true or not One goal may consist of a number of sub-goals and only be solved by solving all sub goals independently Another goal may be solved by solving only some of sub goals This gives rise to a great deal of potential parallelism AND parallelism, OR parallelism and Stream-parallelism.

Efficiency of execution is a major problem In this model of programming Thus, even if a logic programming language might be suitable for a certain application, lack of efficiency may restrict its use primarily to a rapid prototyping tool. The programming languages belonging to this category include PARLOG [18] and Concurrent PROLOG [71]

2.1.3 Functional Computation Model

In imperative programming, the basic element of programming is the assignment statement, which modifies a value of a variable as a result of computation In functional programming, the basic element is a function, which receives input and produces output as a result of function application

A function describes the relationship between input and output The essence of the functional computation is to combine such functions and produce more powerful functions so that a solution for a problem can be obtained by applying such functions to input The functional languages based on this model are referentially transparent and the programmers cannot introduce race conditions As a result, this model has great potential of exploiting implicit parallelism by removing side-effects caused by

assignment statements Functional programming has been one of the main directions in developing new languages that directly address the challenge of parallel programs. In functional languages, computational abstractions are expressed through functions. A first-order function takes data objects as arguments and produces new data objects as results. The function abstracts the method used to produce the new objects from the arguments. High-order functions generalize this further by taking data objects as well as other functions as arguments and producing new data objects and new functions. It can be further advanced by the referential transparency. Because the arguments of a function could be evaluated in any order, functional programming languages promise to be more portable and easily maintained through an entire software life cycle.

However, due to its history insensitivity, the expressive power of programming is limited. This model is not suitable for expressing inherently concurrent natures. The data flow model shares many common properties with the functional model. In the data flow model, data "flows" from one statement to another, execution of statements is data driven, and identifiers obey the single-assignment rule. In a data flow computer, the availability of input operands triggers the execution of the instruction which consumes the inputs. The programming languages belonging to this category include ParAlfl [42], SISAL [60] and Multilisp [38].

2.1.4 Object-Oriented Computation Model

The object object programming model is based on the concept of an object. A program is a set of objects. Each object is a self contained entity with its own private data and a set of methods to manipulate those data. Any access to the data of an object must be done by calling an appropriate method. The behavior of an object is defined by its class, which is a description of a set of objects having a common

behavior An inheritance mechanism allows a class to be defined as an extension of another class

In a traditional object oriented programming model, there is always one active object at a time The active object can send a message to a receiver object Then the receiver object becomes active and the sender object must wait After the receiver object returns a result to the sender object and becomes inactive, the sender object can continue its computation To incorporate parallelism in this model, a number of means have been proposed [9].

1) allow an object to be active without receiving messages.

2) allow the receiving object to continue execution after it returns its result

3) broadcast a message to several objects at once

4) allow the sender of a message to proceed in parallel with the receiver

Any combination of these can be used

An object-oriented programming model naturally reveals existing parallelism in problems [83]. Besides the advantages of modifiability, maintainability and reusability, one advantage of this model over others is that the concept of an object can be used at earlier stages of software development cycles than the implementation stage. It implies that parallel processing aspects such as parallelism and communication among parallel components can be naturally handled at the earlier stage of the software development development Consequently, it is easy for the programmers to handle parallelism and communication among parallel components However, in this model, parallel execution among concurrent objects is the only source of parallelism, and the amount of parallelism to be exploited may not be sufficient for effectively utilizing many fine-grain processors The programming languages belonging to this category include ABCL/1 [89], Act 1 [54], Concurrent Smalltalk [88], Emerald [12] Mentat [36] and POOL [4]

2.1.5 Knowledge-based Model

In addition to those computation models we presented, there are at least three computation models which perform their computation in different ways from the previous models. The rule-based model performs the computation by firing rules. In a production system language, such as OPS5 [14], the basic concepts are IF-THEN statements. These rules are repeatedly executed until none of IF conditions is true. In cellular models, the computation is performed by a set of cells. Each cell performs its computation by communicating with its neighbors in a regular pattern that matches the flow of data and control in the target computation. The class of programming language belonging to this model is a semantic network language, such as NETL [27]. In neural models, the basic concepts are statements defining the network topology together with the recall (and training) functions of the neurons. The execution occurs each time the network is updated. The languages belonging to this group can be regarded as knowledge-based languages and are used for applications in artificial intelligence, cognitive psychology, and learning systems.

2.2 Overview of Parallel Programming Approaches

There are three approaches to programming parallel computers. One approach is to write programs using conventional sequential programming languages, such as Fortran [5, 20, 43] or C [5, 28], and parallelize the programs using parallelizing or vectorizing compilers. Although this approach seems to be attractive since many types of existing sequential software can be adapted to such a parallel programming environment with minor modifications, it is hardly an effective approach. The parallelizing or vectorizing compilers fail to unravel most of the parallelism. They can only detect parallelism associated with iterations over common data structures, such as arrays and matrices [7], and require extensive dependency analysis [3]. Usually, sequential

languages are extended with compiler directives in order to help the compilers detect parallelism. Because these extensions are machine-specific, portability of programs is hampered. Even after extensive research and development, parallelising compilers have not met the expectations that were raised for them. The problem in this approach is not the compilers themselves, but the inherent sequential characteristics of imperative programming languages. As we discussed above, these languages are designed for sequential execution in sequential processors. Although this approach is very popular in scientific application areas, it does not provide promising directions for the future of parallel programming.

Another approach is to use parallel language constructs to explicitly model the parallelism in programs. These parallel language constructs include the parallel statement and *input*, *output* commands in CSP [40], *monitor* and *wait*, *signal* operations in Concurrent Pascal [13], and *task* and *rendezvous* mechanisms in Ada [22]. Although in this approach the imperative languages are extended with some language constructs, the basic modal of computation is still sequential as in the first approach. Using the parallel language constructs, it is the programmers' responsibility to explicitly express the parallelism and ensure the correct communication and synchronization among parallel units, which is an extremely complex and difficult task. This is one of the major flaws in software systems for parallel computing, and there is no easy solution in sight. In addition, these parallel language constructs are only suitable to express coarse grain parallelism. Thus, massive and fine parallelism cannot be expressed in this approach.

The third approach is to use parallel programming languages, such as Id Nouveau [62], and SISAL [60], which are functional languages tailored for scientific computation, PARLOG [19], a parallel logic language, and Act 1 [54], an object-based language based on the Actor model. The underlying computation models of these parallel programming languages are fundamentally different from the underlying models of imperative programming languages in that parallelism is mostly implicit and massive parallelism is easily obtainable. Hence, the programmers using these languages are liberated from the complications caused by parallelism. This is considered the most promising approach to parallel programming, and our computation model belongs to this category. We will discuss it further in the following section.

2.3 Parallel Language Approaches

Sarkar [70] has identified three fundamental issues to be resolved for parallel execution of a program: 1) identifying parallelism in the program, 2) partitioning the program into sequential tasks, and 3) scheduling the tasks on processors. Although these issues have not been treated as a whole, they are closely related in the sense that the choice of a strategy in one area may significantly affect the choice of a strategy in other areas. Sarkar introduced a compile-time method for automatically partitioning data flow graphs. The goal is to find a schedule to minimize the completion time of the program by avoiding exploitation of too fine a grain of parallelism. This approach incorporates three steps: 1) construct the program graph, 2) partition the graph, and 3) generate the target code. In step 1), the execution time is assigned to each node and the communication time between two neighboring nodes is also assigned to the edge connecting the two nodes. In step 2), the program graphs are partitioned. Finally, in step 3), the macro data flow modules are generated. Each macro data flow module is a code segment to be executed sequentially in one processor. This method

has been applied to the SISAL programs, and is regarded as the most sophisticated methods to partition a program. However, this automatic partitioning system can only exploit parallelism appearing in the program graph. Some patterns of parallelism cannot be shown in the program graph, and thus the compiler usually does not have enough information to detect such parallelism and exploit it in a proper way. For instance, pipelined parallelism is almost impossible to exploit in this approach since the compiler itself cannot detect pipelined parallelism. In addition, the parallelism due to the arbitrary producer-consumer relationship cannot be exploited at all.

Goldberg [34] developed a method to programming parallel processors for functional programs by introducing a logical construct called a *serial combinator*. A serial combinator is defined as a function with the following properties: 1) its body contains no free variables, 2) its body is sequential and contains constructs for synchronizing its execution with other tasks; 3) its body could not occur as a subexpression within the body of another serial combinator. In this approach, the third property implies that the programmers have to determine as few serial combinators as possible, since they cannot be coalesced to form a bigger serial combinator. It also implies that the program developed for one parallel computer may not be directly portable due to possible performance degradation. In addition, the programmers must ensure the correct synchronization and communication among tasks.

In [47, 48], Jagannathan et al. introduced an approach to programming a network of workstations based on a coarse-grain dataflow concept. Their computation model incorporates two different paradigms: at the lower level procedural language is used to express functions or procedures, at the high level a declarative style of programming is used to express implicit parallelism among the functions written in procedural programming languages. Foster and Taylor [31] introduced *Strand* for

parallel programming, which is based on logic computation model. Strand can provide an interface to other languages as in [47, 48]. They have ignored the issue of the granularity of parallelism and assume that the programmers will make choices on the grain size during the development of the application. Another approach based on the coarse-grain dataflow computation model was introduced by Gaudiot and Lee [33]. Their approach was aimed at the use of shared memory parallel processors for solving scientific problems.

Although Linda [15] is not based on any specific computation model or approach, it is worth comparing it with other approaches. Linda is a small set of operations that can be added to a base language to create a parallel processing dialect. The concept of Linda is based on the tuple space of parallel processing. Processes and data can be considered to be elements in tuple space. Communication between processes occurs in the following way: the sender creates data in tuple space; the receiver gets the data in the tuple space, hence communication takes place. Linda provides the following four basic operations: *in*, *out*, *rd* and *eval*. *in* removes the tuple read from the tuple space, *rd* reads the tuple, but leaves the tuple to be read by other processes, *out* creates a new tuple and places it in the tuple space, and *eval* creates a new tuple by generating a process. A disadvantage of Linda is that the programmers have to write programs in terms of communication with other processes. In addition, its implementation on distributed memory parallel processors requires significant overhead to support communication via shared memory. Another disadvantage in Linda is that the use of conceptually shared memory does not provide any protection from erroneous access to any data item in the shared memory. When we consider the fact that data or messages need not be accessible to any other processes than those processes which need them, Linda cannot support an information hiding principle.

In conclusion, these existing approaches share some or all of the following disadvantages.

- Software engineering concepts for managing parallelism have not been fully incorporated.
- Most approaches are targeted for the shared memory processors.
- The concept of shared data has not been introduced into programming parallel processors
- Coding of correct synchronization and communication using explicit constructs is still the programmer's responsibility

In order to develop software for parallel processing systems, much effort is still needed to address these problems.

2.4 Contributions of the Dissertation

In developing an approach to software development for parallel processing systems integrating object oriented and functional paradigms, the dissertation makes the following contributions:

- The Parallel Object Oriented Functional Computation Model (PROOF) has been developed. In PROOF, the object-oriented concepts have been integrated to the functional paradigm without sacrificing the advantages of either paradigm. PROOF offers the following advantages: 1) Parallelism in different granularity levels can be exploited. 2) By separating synchronization constraints from the operation of the method, inheritance and parallelism can be integrated without interference. 3) Parallel aspects of the software can be treated as a primary issue at the early stage of the software development so that the complications at the later phases of the software life cycle can be reduced

- The intermediate form, called Intermediate Program Representation (IPR), for the internal representation of the PROOF/L programs has been developed. The IPR is a hybrid graphical representation in which object-level behavior is represented as a Petri net and method-level behavior is represented as function nodes and their data dependency relationships. Correctness preserving transformation rules from the PROOF/L program to the IPR and from the IPR to a target language have also been developed. The two-level transformation via the IPR facilitates the separation of the semantic issues from the performance-oriented issues.

- A two level allocation approach has been developed. At the object level, we model the PROOF program as a directed graph by analyzing the body of each object and then cluster the objects so that each cluster can be analyzed separately at the method level analysis. At the method-level, grain size determination techniques for various types of parallelism have been developed and compared with other existing approaches using examples. In order to analyze the grain sizes, we develop the notion of gain with which the possible contribution to the reduction of the completion time of the program can be determined.

CHAPTER 3
PROOF COMPUTATION MODEL

The object-oriented paradigm is considered promising for the development of software for parallel processing systems [83, 85, 86, 87]. Such a paradigm is based on the principle of information hiding. The main mechanisms associated with the object-oriented paradigm are data encapsulation and inheritance. These mechanisms offer many advantages, such as comprehensibility, reusability, maintainability, modularity and extensibility. In addition, the object-oriented paradigm can naturally reflect the structure of the problem space. In traditional object-oriented computation models, there is always one active object at a time. The active object can send a message to a receiver object. Then the receiver object becomes active and the sender object waits. After the receiver object returns a result back to the sender object and becomes inactive, the sender object can continue its computation.

This model is inherently sequential since it was developed for sequential computers. Some attempts have been made to incorporate parallelism in the object-oriented paradigm. In POOL-T [4], parallelism is achieved by allowing execution of the objects without receiving a message. Concurrent Smalltalk [88], Hybrid [61] and languages based on the Actor model [2], such as Act++ [49] and Rosette [78], introduce parallelism by adopting asynchronous message passing. The actor model supports delegation instead of inheritance. However, in these languages, only parallelism among the objects can be expressed and thus exploitation of fine grain parallelism is difficult or expensive. Our computation model is an approach for exploitation of various levels of granularity of parallelism based on object-oriented and functional paradigms.

In PROOF, a program is represented as a set of the objects which can be executed in parallel. Each object is an instance of a class, and each has its own local data and methods. The methods in PROOF are defined as purely applicative functions. Parallelism at different levels of granularity can be exploited. The major features of PROOF are

- Class, object, and inheritance are supported in PROOF with the restriction that all the methods in the objects are applicative functions. In other words, a functional paradigm is adopted at the level of method definition.
- The *guard* of the method is introduced to support the synchronization between the concurrent objects. Furthermore, it will be shown that the methods along with their guards are inheritable.
- Objects are persistent in PROOF. The *reception of values* by the objects is introduced to modify the objects.

These features are discussed in detail in the following. This chapter is organized as follows. In Sections 3.1 - 3.5, the features of PROOF are presented with examples. In Section 3.6, the sources of parallelism in PROOF and its semantics are presented.

3.1 Classes and Objects

Class interface and definition

In PROOF, a program consists of a set of the objects. Every object is an instance of a class. A class is a template for a set of the objects bearing similar behavior, and it is defined as a *generic abstract data type*. A class is defined by its interface and definition. The class interface describes the types of the methods provided by the class. The class definition consists of the composition of local data and methods, which are purely applicative functions.

```
class interface Bounded_Buffer(itemtype, size)
   method put  : Bounded_Buffer → itemtype → Bounded_Buffer
   method get  : Bounded_Buffer → Bounded_Buffer x itemtype
   method is_empty : Bounded_Buffer → bool
   method length : Bounded_Buffer → int
end class
```

Figure 3.1 The interface of the class Bounded_Buffer

It is important to point out that restricting the methods of classes applicative functions does not restrict the expressive power of PROOF, but merely requires that all the effects of the methods be explicitly specified via the parameters of the methods.

Example 1. Class interface and definition of Bounded_Buffer

We introduce the Bounded_Buffer example to illustrate the features of PROOF. The class Bounded_Buffer is a FIFO queue of a limited capacity. The class interface and the definition of Bounded_Buffer are shown in Figures 3.1, and 3.2. Bounded_Buffer has two parameters: itemtype and size. itemtype indicates the type of elements to be stored in the buffer and size specifies the capacity of the buffer. The method names are self explanatory. In the class definition, the **composition** clause defines the composition of the local data of the class. The composition of Bounded_Buffer is a Cartesian product. The components can be referred using the dot syntax, such as b.store.

Inheritance and genericity

Both inheritance and genericity are supported in PROOF. Classes in PROOF are related by *inheritance* relations. Inheritance is used to define a subclass as a specialization of a superclass. In a subclass, all the local data and the methods of its superclass are inherited. Additional local data and new methods may be introduced. The inherited methods may also be overridden by a new definition of the method.

```
class definition Bounded_Buffer(itemtype, size)
   composition
      store list(itemtype) × count:int
   method put b x
      expression
         β[(sppend_right x), inc] [b]
   method get b
      expression
         [ β[tail,dec] [b], head(b store) ]
   method is_empty b
      expression
         b.count = 0
   method length b
      expression
         b.count
end class
```

Functions used here are defined as follows.

$(\text{sppend\_right } x)\ y = y \circ x$

$\text{tail}[e_1, e_2, \ldots, e_n] \equiv [e_2, e_3, \ldots, e_n]$

$\text{head}[e_1, e_2, \ldots, e_n] \equiv e_1$

$\text{inc}(n) \equiv n + 1$

$\text{dec}(n) \equiv n - 1$

$\beta[f_1, \ldots, f_n][x_1, \ldots, x_n] \equiv [f_1(x_1), \ldots, f_n(x_n)]$

Figure 3.2. The definition of the class Bounded_Buffer without guard constructs

Example 2. Inheritance of classes.

The class interface and definition of Extended_Buffer, which is a subclass of Bounded_Buffer is shown in Figure 3.3. In Extended_Buffer a new LIFO method, pop, is added, and all the methods in Bounded_Buffer are inherited. □

Genericity is used to define parametric or generic classes. An instantiation of a generic class is obtained by assigning values to the parameters of the generic class. All the local data and the methods of the generic class are inherited. The

```
class interface Extended_Buffer(itemtype, size)
   inherit put, get, is_empty, length from Bounded_Buffer
   method pop   Extended_Buffer → Extended_Buffer × itemtype
end class

class definition Extended_Buffer(itemtype, size)
   superclass. Bounded_Buffer(itemtype, size)
   inherit put, get, is_empty, length
   method pop b
      expression
         [ β[front,dec] b, last(b store) ]
end class
```

Figure 3 3 The definition of the class Extended_Buffer

Bounded_Buffer in Example 1 is a generic class An instantiation of Bounded_Buffer is the following

```
class My-Buffer instance of Bounded_Buffer(my-item, my-size)
```

Class My-Buffer is an instantiation of the generic class Bounded_Buffer

Active and passive objects

A program in PROOF consists of a set of the objects. The objects can be classified into the following three categories *active, passive* and *pseudo-active.*

Definition 3.1.1 The active *object is an object which can initiate a process by invoking a method or methods without any request from the other object.*

Definition 3.1.2 The passive *object is an object which has methods to be invoked by the other objects and cannot invoke them without a request from other objects*

Definition 3.1.3 The pseudo-active *object is an object which cannot initiate a process but can invoke the methods defined in another object upon request by other objects*

```
class interface Producer_Class(itemtype)
    method produce → itemtype
end class

class interface Consumer_Class(itemtype)
    method consume itemtype →
end class
```

Figure 3 4 Interfaces of the classes Consumer_Class and Producer_Class

Definition 3.1.4 *The* non passive *object is an object which is not passive, and thus includes* active *and* pseudo-active *objects*

Definition 3.1.5 *The* non active *object is an object which is not active, and thus includes* pseudo-active *and* passive *objects*

A non-active object acts like a service agency It waits passively until one of its methods is invoked by other objects The non active object may in turn invoke methods in other objects. An active object is active initially, and it may remain active throughout the execution except for occasional suspensions for the purpose of synchronization with other objects A *body* will be attached to each non passive object The bodies of the objects are functions that may be recursive and diverse (non terminating)

Example 3 Producer consumer problem

Interfaces of two classes, Consumer_Class and Producer_Class, are shown in Figure 3 4 The method produce() is to generate an item every time it is called, and the method consume() is to consume an item every time it is called The objects in the producer-consumer problem are shown in Figure 3 5. Producer and Consumer are active objects. Producer will continuously generate items and Consumer will

```
Object Buffer   instance of Bounded_Buffer (my-item, my-size)

Active Object   Producer  instance of Producer_Class (my-item)
       Body     /* a body is attached to Producer*/

Active Object   Consumer  instance of Consumer_Class (my-item)
       Body     /* a body is attached to Consumer*/
```

Figure 3.5 The set of the objects for a producer consumer problem

continuously consume items. The definition of the bodies of Producer and Consumer will be given later. Buffer is a passive object, and it becomes active when its methods are invoked by Producer or Consumer.

3.2. Method Definition

Methods in PROOF are purely applicative functions or functional forms, i.e. high order functions. We use a constructor $[x_1, x_2, \ldots, x_n]$ to denote a sequence of homogeneous or heterogeneous elements. In the case of homogeneous elements, it denotes a *list* or an *array* whose types are T^* and $T^n (\equiv \underbrace{T \times T \times \ldots \times T}_{n})$ respectively. In the case of heterogeneous elements, it denotes a *Cartesian product* whose type is $\prod_{i=1}^{n} T_i \ (\equiv T_1 \times T_2 \times \ldots \times T_n)$.

PROOF supports a set of primary functions, such as arithmetic operations, logic operations and list handling operations and functional forms from which other functions and functional forms can be easily constructed. The following are the functional forms currently supported in PROOF.

a) Functional form: α (called *apply to all*)

$$\alpha f[x_1, x_2, \ldots, x_n] \equiv [f(x_1), f(x_2), \ldots, f(x_n)]$$

α has two parameters, a function of type $T_1 \to T_2$, and a list of homogeneous elements of type T_1. The function f is applied to each element in the list and yields a list of elements of type T_2.

b) Functional form β (called *distributed apply*)

$$\beta[f_1, f_2, \ldots, f_n][x_1, x_2, \ldots, x_n] \equiv [f_1(x_1), f_2(x_2), \ldots, f_n(x_n)]$$

β has two parameters, a list of functions in which each function f_i is of type $T_i^{(1)} \to T_i^{(2)}$, and a list of heterogeneous elements in which the ith element is of type $T_i^{(1)}$. Each function in the first list is applied to the corresponding element in the second list. It yields a list in which the ith element is of type $T_i^{(2)}$.

c) Functional form γ (called *filter*)

$$\gamma[b_1, b_2, \ldots, b_n][x_1, x_2, \ldots, x_n] = \begin{cases} [\,] & \text{if } n = 1 \text{ and } b_1 = \texttt{False} \\ [x_1] & \text{if } n = 1 \text{ and } b_1 = \texttt{True} \\ \gamma[b_1 \ldots b_k][x_1 \ldots x_k] \circ \gamma[b_{k+1} \ldots b_n][x_{k+1}, \ldots x_n] & \text{if } n > 1 \text{ and } 2 \le k \le n \end{cases}$$

Here, $\circ$ denotes the concatenation of two lists. γ has two parameters, a list of boolean and a list of any elements. This function yields a subsequence of the second list by selecting elements whose corresponding elements in the first list are **True**.

d) Functional form δ (called *copied apply*)

$$\delta[f_1, f_2, \ldots, f_n]x \equiv [f_1(x), f_2(x), \ldots, f_n(x)]$$

δ has two parameters, a list of functions in which each function f_i is of type $T \to T_i$, and an element is of type T. Each function in the first list is applied to the element and a list of elements having type $T_1 \times T_2 \times \ldots T_n$ is yielded as a result of δ application.

e) Functional form η (called *inserted apply*)

$$\eta[f][x_1, x_2, \ldots, x_n]$$
$$\equiv [f(x_1, f(x_2, \ldots, f(x_{n-2}, f(x_{n-1}, x_n)) \quad)]$$

η has two parameters, a function of which type is $T \to T$ and a list of elements in which each element is of type T. The function is first applied to the last two elements. Then the result of that application and the third from the end of the list are used as input to another function application. This application is continued until the first element of the list is input to the function. In fact, this functional form can be given a different meaning by allowing parallel application of the function to the elements. That is, we can group the elements into a set of pairs and apply the function f. By repeating this process until a value is returned as a result, we can exploit tree parallelism.

The following example illustrates the parallelism due to parallel evaluation of arguments and functions.

Example 4. Partition an array of integers, A which has type int^n, using a pivot element $s \in A$ as shown in Figure 3.6. In other words, we want to rearrange the elements in the array A so that all the elements whose values are less than the value of s precede the elements whose values are greater than or equal to s.

A solution to the problem can be formulated as follows. First, we define the following two functions f and g.

f : $\text{int} \to \text{bool}$

$f\, x \equiv x < s$

g : $\text{int} \to \text{bool}$

$g\, x \equiv x \geq s$

One solution to the problem is the following one-liner:

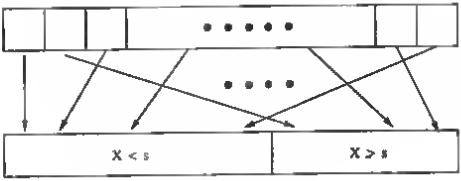


Figure 3.6. An array partitioning problem

$$(\gamma\ (\alpha\ f\ A)\ A\ ) \equiv (\gamma\ (\alpha\ g\ A)\ A\ )$$

The functions are curried and we assume left association. This expression is a concatenation of two γ functions, which can be evaluated in parallel. Within each of the γ functions, the applications of f and g to A can also be evaluated in parallel owing to the functional form α. The execution of this example is illustrated in Figure 3.7.

□

3.3 Synchronization of Objects

The methods defined in each object may require some preconditions under which they can be executed. For example, `get` in `Bounded_Buffer` can be executed only when the buffer is not empty. When the buffer is empty, the invoker of the method must be suspended. This problem is commonly known as a synchronization problem.

In order to specify the synchronization constraints in the parallel object-oriented paradigms, many constructs have been proposed. Among them are critical sections [88], message queues [61], behavior abstraction [43] and enabled-sets [78]. However,

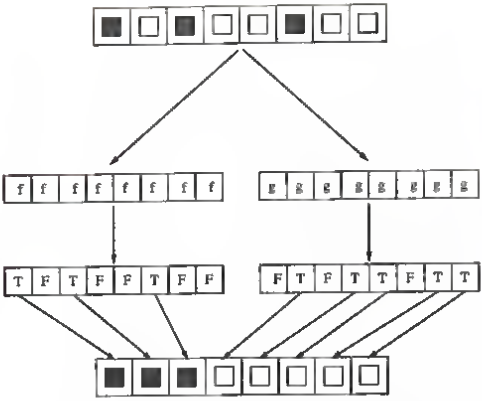


Figure 3.7 A solution for the array partitioning

the use of these constructs interferes with the inheritance mechanism, and the synchronization constraints are not directly inheritable with the methods. In PROOF, synchronization among the objects is achieved by attaching an optional precondition, called *guard* [23], to each of the methods in a class. Each guard is a predicate. The object which invokes the method is suspended when the attached guard evaluates to `False`, and it is resumed when the guard becomes `True`. The guard attached to the method is defined in a way that it depends only on the status of the local data, and does not on the definition of any other methods. Therefore the inheritance of

```
class definition Bounded_Buffer(itemtype, size)
   composition store list(itemtype) × count int
   method put b x
      guard(b count < size)
      expression
         β[(append_right x), inc] b
   method get b
      guard(b count > 0)
      expression
         [ β[tail,dec] b, head(b store) ]
   method is empty b
      expression
         b count = 0
   method length
      expression
         b count
end class
```

Figure 3.8 The definition of the class Bounded_Buffer

individual methods will not be hampered by the inclusion of the guard. Such guards can be inherited with the methods they are attached to.

Example 5 Inheritance of synchronization constraints

The complete definition of the class Bounded_Buffer with guards is shown in Figure 3.8. The guard attached to the method get, (b count > 0), indicates that get can be executed only when the buffer is not empty.

The guard attached to put, (b count < size), indicates that put can be executed only when the buffer is not full. The complete definition of Extended_Buffer, which is a subclass of Bounded_Buffer, is shown in Figure 3.9. The Extended_Buffer inherits the methods including the guards of the Bounded_Buffer. The inherited guards work coherently with the new guard defined in pop. □

```
class definition Extended_Buffer(itemtype, size)
   superclass Bounded_Buffer(itemtype, size)
   inherit put, get, is empty, length
   method pop b
      guard(b count > 0)
      expression
         [ β[front,dec] b, last(b store) ]
end class
```

Figure 3.9 The complete definition of the class Extended_Buffer

This example has been used to illustrate the difficulty in integrating inheritance and synchronization constraints in some object oriented parallel models [49, 78]. The above example demonstrates inheritance can be integrated with synchronization constraints without difficulty in PROOF. By separating synchronization constraints (guard) from the behavior of methods (expression), integration of inheritance and parallelism can be achieved with interference. Only modification of the bodies is required.

Since each method is associated with a guard specifying the condition under which the method can be executed, each method can be a natural unit of inheritance.

3.4 Persistence of Objects

A major deficiency of the functional paradigm is its history-insensitivity. PROOF is made history sensitive by making the objects persistent and allowing the reception of values by the objects, i.e. the assignment of values to the objects. The local data of the objects is persistent. The reception of values by the objects will modify the local data of the objects. A pseudo-function $\mathcal{R}$, called the *reception function*, is introduced to denote the reception of a value by an object.

$$\mathcal{R}\,[\,o\,]\,(e)$$

```
Object Buffer   instance of Bounded_Buffer (my-item, my-size)

Active Object   Producer : instance of Producer_Class (my-item)
     Body       while True ( R [Buffer] (put Buffer produce))

Active Object   Consumer : instance of Consumer_Class (my-item)
     Body       while True ( β[R [Buffer], consume] (get Buffer))
```

Figure 3.10 The definition of the objects for producer-consumer problem

$\mathcal{R}$ is not a function, but can be treated as such. $\mathcal{R}$ has two parameters: an object o, the recipient, and the expression e, to be received by o. e may contain applications of applicative functions only. This pseudo-function can only appear inside bodies of active objects, and may not be nested. Major differences between modification of the objects through $\mathcal{R}$ and traditional assignments are:

- The evaluation of the expression e in $\mathcal{R}$ can be done in parallel, since e contains only applications of purely applicative functions.
- No partial modification to the object o is allowed. The local data of an object can only be modified as a whole entity, i.e. its components cannot be modified individually.

The restriction that the reception function can only appear in the bodies of the objects implies that all the methods in an object are still applicative, i.e. there are no side-effects.

This restriction effectively preserves the referential transparency at the method level, and retains the parallelism owing to that referential transparency.

Example 6 Bodies of active objects

The complete definition of the objects for a producer-consumer problem is shown in Figure 3.10. In the body of `Producer`, `put` adds an item produced by the method

produce to the buffer, and returns a new buffer as its result. The object `Buffer` will receive the value of the new buffer. In the body of `Consumer.get` returns two values a new buffer with an item deleted from the original buffer and the deleted item. The `Buffer` will receive the value of the new buffer and the deleted item will be consumed by the `Consumer` as the result of invoking `consume`. □

Simultaneous access to the objects

An object can participate in more than one function evaluation. There may be several attempts to modify the same objects at the same time. Simultaneous modification of the objects may result in inconsistent and incorrect states. Thus, simultaneous modification to the same object must be serialized so that at any given moment an object will be a recipient of only one of the function evaluations. In order to illustrate simultaneous access to the objects, let's consider the methods defined in the `Bounded_Buffer` shown in Fig 3.8. Simultaneous invocations of `length` and `put`, or `length` and `get` are permitted since only one of the methods will attempt to modify the object. On the other hand, simultaneous invocation of `put` and `get` are prohibited since both the methods will attempt to modify the object. Thus, they must be serialized.

At any moment, the status of any object involved in an expression falls into one of the following three categories:

read only : The expression only needs to read the value of the object.

will modify : The expression will modify the object, but the modification does not occur at this moment.

modifying : The expression is currently modifying the object.

In order to ensure the consistency and the correctness of the objects, a multi-mode locking mechanism is adopted. There are three different types of locks, R-Lock,

Table 3.1 Multi mode locking mechanism

	R-Lock	W-Lock	M-Lock
R-Lock	compatible	compatible	incompatible
W-Lock	compatible	incompatible	incompatible
M-Lock	incompatible	incompatible	incompatible

W-Lock and **M-Lock**, that are associated with the three status of the object, **read-only**, **will-modify** and **modifying**, respectively. Before evaluating an expression involving an object *o*, a proper lock for *o* must be obtained. A lock is granted only when it is compatible with other locks granted for the same object, according to the compatibility chart in Table 3.1.

3.5 Control Functions

The functional forms used to express parallelism are based on data dependency. In other words, the application of functions can be ordered according to the input and output relations. However, we may need functions whose relations are not directly based on input output relations. For instance, consider a *dining philosophers* problem. In this problem each philosopher either thinks or eats noodles. In order for a philosopher to eat noodles, he first has to acquire two chopsticks. A philosopher repeats a sequence of activities: *think*, *acquire chopsticks*, *eat noodles* and *release chopsticks*. If we consider each activity to be a function, each can be designed as a method in PROOF. In this case, there is no significant data flow between *think* and *acquire chopsticks*. As the programmer knows from the requirements of this problem that every philosopher will first *think* and then request chopsticks by invoking *acquire chopsticks*, he has to express such sequential ordering of activities. However, the function *think* may not have significant output generated and used as input to the function *acquire chopsticks*. Thus, if we do not have a *sequential* control function,

we need to create data flow artificially between these two functions In fact, we can use a kind of *flag* as an output parameter of a preceding function to indicate the completion of the preceding function, such as *think* The same *flag* is used as an input parameter to trigger the following function *acquire chopsticks* The use of such *flags* not only makes programming very complicated, but also reduces readability and modifiability of the programs

To alleviate such problems, we introduce the following functions, called *control* functions Since there are applications which require explicit control structure other than data flow, we need additional constructs to express high level *sequential* or *parallel* control structures ; is a *sequential* control function defined as follows

a) Control function ; (called *sequential*)

$$;[e_1, e_2, \ldots, e_n] \equiv e_1 ; e_2, \ldots e_n$$

in which , means sequential execution It is not necessary that there are any data dependency relations among e_is

// is a *parallel* constructs defined as follows

b) Control function // (called *parallel*)

$$//(e_1, e_2, \ldots, e_n) \equiv e_1 // e_2 // \quad // e_n$$

in which // means parallel execution These control functions are used to specify sequential control flow or explicit parallel control flow among functions in the bodies of the objects.

In addition to these explicit control functions, we also define *while* and *if* control functions as follows:

e) Control function *if* (called *conditional select*)

$$\textit{if}\ [p\ f\ g][x] = \begin{cases} f(x) & \text{if } p(x) \text{ is true} \\ g(x) & \text{if } p(x) \text{ is false} \end{cases}$$

in which p, f and g are of type $T \rightarrow \textit{boolean}$, $T \rightarrow T_f$, and $T \rightarrow T_g$, respectively, and x is of type T

d) Control function *while* (called *conditional loop*)

$$\textit{while}\ [p\ f][x] = \begin{cases} \textit{while}[p\ f][f(x)] & \text{if } p(x) \text{ is true} \\ x & \text{if } p(x) \text{ is false} \end{cases}$$

in which p and f are of type $T \rightarrow \textit{boolean}$, $T \rightarrow T$, respectively, and x is of type T The control functions, *if* and *while* are applicative functions without side-effects They can be used in the method definition as well as in the body

3.6 Parallelism in PROOF

In this section, we discuss the parallelism offered by PROOF and its semantics.

3.6.1 Sources of Parallelism

PROOF offers parallelism primarily in two different levels the object level and the method level The object level parallelism is achieved by allowing more than one object to be active at a time, and the method level parallelism can be achieved by defining methods as applicative functions

Each of the active objects can invoke a method, thus there can be a number of methods being executed simultaneously The potential for the simultaneous participation of an object in the evaluation of different functions also implies another

source of parallelism in PROOF. Method level parallelism is obtained from the following two sources: parallel evaluation of arguments of functions and parallel evaluation of functions or replication of a function. The former is achieved because of *referential transparency* of applicative functions. The latter is achieved by functional forms, such as α and β. In the following, we summarize the sources of parallelism in the computation model. Because all the functions in PROOF are applicative, parallelism is obtained from the following two sources:

P1 Parallel evaluation of arguments of functions. It is made possible because of the *referential transparency* of applicative functions.

P2 Parallel evaluation of a number of functions or replications of a function. It is made possible by functional forms such as α and β.

The active objects introduce another source of parallelism in PROOF:

P3 There can be a number of the objects that are active simultaneously throughout the execution.

P4 An object can be involved in two or more function evaluations simultaneously.

3.6.2 The Parallel Semantics of PROOF

In this section, we will define the semantics of PROOF in terms of its parallel behavior. We give the semantics to PROOF at three different levels: one is at the method definition level (applicative function), another is at the object definition level and the other is the interface level between the two levels.

Parallel evaluation of arguments and parallel evaluation of a number of functions are related to the issue of the semantics of functional programming languages. At the method definition level, an important concern is how to select the expression to be

executed next. There are three different semantics for programs written with functional programming languages, according to how an expression to be executed next is chosen: strict, lazy and lenient. Strict semantics imposes the following requirement during evaluation of the expression: complete evaluation of all arguments before a function call and complete evaluation of all components before a structure is built. This requirement implies that strict semantics may require useless computation that is not necessarily to be executed. This has two disadvantages: (1) the consumption of resources and (2) a failure to produce a solution. It also requires additional effort to implement in distributed memory parallel computers, since some of the unnecessary results may complicate their handling and increase overhead in communication costs.

In contrast to such strict semantics, nonstrict semantics does not impose any requirement mentioned above. Both lazy and lenient semantics are nonstrict in the sense that any of the above requirements for strict semantics is not required. Instead, lazy semantics imposes the following requirement: a computation cannot be executed unless it contributes to the final answer. This ensures that function arguments, structure components and arms of conditionals are not evaluated unless they are needed. Furthermore, termination of the computation and obtaining the final answer occurs at the same time, since there will be no further reductions required when a normal form is reached. As a consequence, unlike the case of strict semantics, lazy semantics guarantees minimal computation to obtain a final answer if it exists. However, the lazy semantics may require a substantial amount of work to determine the next computation to be executed.

Lenient semantics [79] imposes the following requirement: complete evaluation of the predicate before any arm of a conditional is considered for execution. Thus, unnecessary computations are avoided only when they are guarded by appropriate conditions. For instance, consider the control function *if* [*p f g*] [*x*]. Under lenient

semantics, first $p(z)$ will be evaluated. Then, based on the result of the evaluation, either $f(z)$ or $g(z)$ will be chosen to execute. When the expression is executed based on strict semantics, the three subexpressions, $p(z)$, $f(z)$ and $g(z)$, can be executed in parallel, since lenient semantics can be considered as a compromise between strict semantics and lazy semantics. Lenient semantics can eliminate substantial effort in determining the next computation to execute. It can also initiate many computations, and thus increase the opportunities for parallel execution. In PROOF we adopt the lenient semantics due to the advantages for parallel processing. In the following, we give the semantics of PROOF in its parallel aspects. The key element in the definition of the semantics is the *parallelizer* denoted Para, which describes how a program in PROOF can be executed in parallel. The parallelizer Para is defined using four evaluation schemata presented in the following order:

Schema A: evaluation of applicative functions

Schema B: evaluation of the pseudo-function $\mathcal{R}$

Schema C: evaluation of control functions

Schema D: evaluation of a set of the objects which composes a program in PROOF

It is shown that the parallel execution of a program according to the parallelizer Para is equivalent to a sequential execution of the same program.

Schema A Let e be an expression consisting of applications of applicative functions only. Para(e) is defined as follows:

Case 1: e is a constant.

Para$(e) \equiv e$

Case 2: e is a function $f(e_1, e_2, \ldots, e_n)$ except the functions *if* and *while*.

Para$(f(e_1, e_2, \ldots, e_n)) \equiv$

parbegin

$v_1 \leftarrow$ **Para** $(e_1) \| v_2 \leftarrow$ **Para** $(e_2) \| \ldots \| v_n \leftarrow$ **Para** (e_n)

parend

$v \leftarrow f(v_1, v_2, \ldots v_n)$

return v

Case 3: e is a function *if* $[p\ f\ g][x]$

Para$(if \mid p\ f\ g\ \mid [x]) \equiv$

if Para$(p(x))$ then

$v \leftarrow$ **Para**$(\ f(x)\ )$

else

$v \leftarrow$ **Para**$(\ g(x)\ )$

return v

Case 4: e is a function *while* $[p\ f][x]$

Para$(while \mid p\ f\ \mid [x]) \equiv$

if Para$(p(x))$ then

while $[p\ f\ \mid](f(x))$

else

return x

Here, **parbegin** ... $\|$ ... $\|$ ... **parend** is used to indicate the parallel execution of elements separated by $\|$. The Case 3 shows that PROOF supports the lement semantics by selecting only one arm for execution after evaluating a conditional. If we treat the *if* control function as a general function and apply it to the semantics in the case 2, the strict semantics can be obtained. We assume that **Para**(e_i) terminates[1]. However, in real execution e_i may not terminate. In such a case, we can use a kind of *fair* scheduling strategy. That is, we can prevent such non-terminating process from

[1] We can easily extend this to non-strict semantics by evaluating f in parallel with **Para**(e_i)

exhausting the system resources by limiting service time for each process. With this provision, lenient semantics can produce the same answers as lazy semantics with only a bounded amount of excess computation.

Variations of schema A can be derived for the functional forms α, β, γ, δ and η according to their definitions.

Scheme A_α: Let e be an expression $\alpha f[x_1, x_2, \cdots, x_n]$

Para$(\alpha f[x_1, x_2, \cdots, x_n]) \equiv$

parbegin

$v_1 \leftarrow$ **Para**$(f(x_1))\|v_2 \leftarrow$ **Para**$(f(x_2))\|$

$\cdots\|v_n \leftarrow$ **Para**$(f(x_n))$

parend

return $[v_1, v_2, \cdots, v_n]$

Scheme A_β: Let e be an expression $\beta[f_1, f_2, \cdots, f_n][x_1, x_2, \cdots, x_n]$

Para$(\beta[f_1, f_2, \cdots, f_n][x_1, x_2, \cdots, x_n]) \equiv$

parbegin

$v_1 \leftarrow$ **Para**$(f_1(x_1))\|v_2 \leftarrow$ **Para**$(f_2(x_2))\|$

$\cdots\|v_n \leftarrow$ **Para**$(f_n(x_n))$

parend

return $[v_1, v_2, \cdots, v_n]$

Scheme A_γ is omitted since it can be considered as a special form of A_β.

Scheme A_δ: Let e be an expression $\delta[f_1, f_2, \cdots, f_n][x]$

Para$(\delta[f_1, f_2, \cdots, f_n][x]) \equiv$

parbegin

$v_1 \leftarrow$ **Para**$(f_1(x))\|v_2 \leftarrow$ **Para**$(f_2(x))\|$

$$\|v_n \leftarrow \mathtt{Para}(f_n(x))$$

parend

return $[v_1, v_2, \ldots, v_n]$

Schema A_η Let ε be an expression $\eta[f][x_1, x_2, \ldots, x_n]$

$\mathtt{Para}(\eta[f][x_1, x_2, \ldots, x_n]) \equiv$

$v \leftarrow \mathtt{Para}(f(x_1), \mathtt{Para}(f(x_2, \ldots, \mathtt{Para}(f(x_{n-2}, \mathtt{Para}(f(x_{n-1}, \mathtt{Para}(f(x_n)) \quad)$

return v

In the following, we prove the schemata to be correct The correctness of schemata $A, A_\alpha, A_\beta, A_\gamma, A_\delta$ and A_η is justified in the following theorem and lemma.

Theorem 3.6.1 The evaluation of applicative functions in parallel described in scheme A is equivalent to a sequential evaluation of the same functions

Proof of Theorem 3.6.1 Since ε consists of applications of applicative functions only, there is no side-effect. The evaluation order of the components(subexpressions) of a function does not affect the result of the function (Church-Rosser theorem [18]) Therefore, it implies that the parallel evaluation of all the subexpressions will yield the same result as the sequential evaluation of the subexpressions in any order

This theorem shows that the evaluation of applicative functions is equivalent to a sequential evaluation of the same program

Lemma 3.6.1 The evaluation of applicative functions in parallel described in schemata A_α, A_β, A_γ, A_δ and A_η is equivalent to a sequential evaluation of the same functions

Proof of Lemma 3.6.1 Since all the functional forms belong to the applicative functions, by Theorem 3.6.1 it is proved

PROOF exploits coarse grain parallelism by allowing more than one object to be active at a time. Unlike the evaluation of applicative functions, parallel execution of the objects involves parallel modifications, which must be controlled in such a manner that only the results equivalent to a series of sequential modifications of the objects are permissible. The locking mechanism used in the concurrency control of data base systems is adopted here to ensure the serializability [63] of parallel modifications of the objects. R-Lock is used to indicate that an expression only needs to read an object (read-only). W-Lock is used to indicate that an expression will modify an object, but not now (will-modify). M-Lock indicates that an expression is currently modifying an object (modifying). The compatibility between these locking modes are shown in Table 3.1.

The modification of the objects is achieved by the pseudo-function $\mathcal{R}$. The execution of $\mathcal{R}[\![\ o\ ]\!](e)$ is described in schema B. First, the following notation is introduced. Let e be an expression.

$$obj(e) = \begin{cases} \emptyset & \text{if } e \text{ is a constant} \\ e & \text{if } e \text{ is an object name} \\ \bigcup_{i=1}^{n} obj(e_i) & \text{if } e = f(e_1, e_2, \cdots, e_n) \end{cases}$$

Schema B Let o be an object name and e be an expression. Para $(\mathcal{R}[\![\ o\ ]\!](e))$ is defined as follows.

Case 1: e is a constant.

Para $(\mathcal{R}[\![\ o\ ]\!](e)) \equiv$

B1.1 M-Lock($\{o\}$)

B1.2 $o \leftarrow e$

B1.3 Unlock($\{o\}$)

Case 2: e is an expression.

Para $(\mathcal{R}[\![\ o\ ]\!](e)) \equiv$

B2.1 repeat

```
B2 2    W-Lock({o})
B2 3    R-Lock(obj(e) - {o})
B2 4    evaluate guards associated with the methods in e
B2 5    if one or more guards are False
B2 6        Unlock(obj(e) ∪ {o})
B2 7    until all guards are True
B2 8    t ← Para(e)
B2 9    M-Lock({o})
B2 10   Unlock(obj(e) - {o})
B2 11   o ← t
B2 12   Unlock({o})
```

In case 1, we only need to request an M-Lock for the object o before the modification, and unlock it after the modification. In case 2, an R-Lock is requested for each object involved in the expression e except the object o, for which a W-Lock is requested. Then all the guards associated with the methods in e are evaluated. If not all of the guards evaluate to True, we unlock all the objects and repeat this evaluation until all the guards are True. Then, after evaluating the expression e, we request an M-Lock for an object o and modify the object o. Finally we unlock the object o.

Theorem 3.6.2 Scheme B ensures that the parallel modification of the objects is equivalent to a sequential modification of the objects, i. e. the parallel modification of the objects is serializable.

Proof of Theorem 3.6.2 In [26], a sufficient condition for serializability is given: the two-phase locking protocol. The two phase locking protocol requires that the locking process consists of two phases, growing and shrinking phases. In the growing phase, locks are obtained, but cannot be released until we complete this phase. In shrinking

phase, locks are released, and no locks can be obtained, including upgrading of locks. In other words, no lock can be obtained after we start to release locks. Therefore we only need to show that schema B satisfies the two-phase locking protocol.

Case 1: *When e is a constant, only one lock is involved. Thus, it is obvious that it satisfies the two-phase locking protocol.*

Case 2: *When e is an expression in which one or more objects are involved, the growing phase consists of B2.2 to B2.9, and the shrinking phase consists of B2.10 to B2.12. In fact, the* repeat *statement in the growing phase may require unlocking all of the objects. However, the two-phase locking protocol is not violated, since once the objects are unlocked, we will not enter the modifying region (B2.10 to B2.12). When we eventually enter the modifying region, we will not request any lock after we start to release locks. Thus, it satisfies the two-phase locking protocol. Therefore, the parallel modification of the objects in PROOF is equivalent to the sequential modification of the objects.*

We give the semantics to the explicit control functions in the following.

Schema C: Let e be an expression $,(e_1, e_2, \ldots, e_n)$

```
Para(,(e1, e2, ..., en)) ≡
    seqbegin
        v1 ← Para(e1),
        return v1,
        v2 ← Para(e2),
        return v2,
        .
        .
        .
        vn ← Para(en),
```

```
            return v_n
        seqend
```

Here, seqbegin , , , , seqend is used to indicate the sequential execution of elements separated by a comma.

Scheme $C_{//}$: Let e be an expression $//(e_1, e_2, \ldots, e_n)$

```
Para(//(e_1, e_2, ..., e_n)) ≡
    parbegin
        seqbegin
            v_1 ← Para(e_1),
            return v_1
        seqend
        seqbegin
            v_2 ← Para(e_2),
            return v_2
        seqend
            .
            .
        seqbegin
            v_n ← Para(e_n),
            return v_n
        seqend
    parend
```

Note the difference between Schema A_g and Scheme $C_{//}$. In the latter, it is not necessary to wait for all the results. However, in the former, the semantics requires collection of the results before proceeding its computation.

Theorem 3.6.3 The evaluation of the explicit control functions described in schemata $C_{;}$ *and* $C_{//}$ *is equivalent to a sequential evaluation of the same functions*

Proof of Theorem 3.6.3 In the case of the sequential control function ; the proof is trivial In the case of the parallel control function //, let a body of an object o, body($\bar{s}$) be an expression //($e_1, e_2, \ldots, e_n$) When the body($\bar{s}$) includes only applicative functions, by Theorem 3.6.1 the parallel evaluation is equivalent to a sequential evaluation When the body($\bar{s}$) includes the pseudo-function or the pseudo-functions, the parallel evaluation is equivalent to a sequential evaluation Therefore, the evaluation of the control functions ; and // in parallel is the equivalent of a sequential evaluation of those functions

Note that Theorem 3.6.3 does not guarantee that the parallel evaluation of the body always gives the same result. Theorem 3.6.3 ensures that it is always possible to find a sequential evaluation which is equivalent to a parallel evaluation of the body For instance, when the body involves more than one modification statement to the same object, i.e. pseudo-function $\mathcal{R}$ whose recipient is the same object, the result varies depending on the execution order of the pseudo-function statements In order to ensure that the parallel evaluation of the body always yields to the same result, we can make a restriction on the definition of the body

Definition 3.6.1 A body is called safe if and only if there is at most one modification statement for each object involved in the body

If a body of an object is safe, all the expressions in that body can be executed in parallel. Thus, parallel evaluation guarantees the same result all the time

Lemma 3.6.2 The parallel evaluation of a safe body ensures a unique result which is equivalent to a sequential evaluation of that body

Scheme D describes the execution of a set of the objects, that compose a program in PROOF.

Scheme D Let **O** be a set of the objects which compose a program, and $\overline{\mathbf{O}} = \{\overline{o}_1, \overline{o}_2, \ldots, \overline{o}_n\}$ be a subset of the objects that are non passive. We use $body(\overline{o}_i)$ to denote the body of a non passive object $\overline{o}_i$. Note that $body(\overline{o}_i)$ is a function which may contain the pseudo-function $\mathcal{R}$. The parallel execution of **O** is defined as follows:

$Para(\mathbf{O}) \equiv$

parbegin

$Para(body(\overline{o}_1)) \| Para(body(\overline{o}_1)) \| \ldots \| Para(body(\overline{o}_n))$

parend

Theorem 3.6.4 Scheme D ensures that the parallel execution of a program in PROOF is equivalent to the sequential execution of the program.

Proof of Theorem 3.6.4 According to Theorem 3.6.1, any evaluation of the applicative functions in parallel is equivalent to the sequential evaluation of the same functions. According to Theorem 3.6.2, any evaluation of the pseudo-function $\mathcal{R}$ in parallel is serializable. According to Theorem 3.6.3, any evaluation of a body is equivalent to a sequential evaluation of the body. Therefore, the parallel modifications involved in the set of the bodies is also serializable.

CHAPTER 4
INTERMEDIATE PROGRAM REPRESENTATION FOR PROOF/L

There are several intermediate languages for functional language implementation, such as IF1 [73], P-TAC [6], and Lean [10]. IF1, which was developed for the implementation of the functional language SISAL, is a hierarchical graph language that describes the dataflow graphs produced from SISAL functions. P-TAC is an intermediate language designed to capture the sharing of computation so that optimization of ID programs can be unambiguously understood and classified during the compilation of ID programs. Lean, which is an intermediate language for specifying computation in terms of graph rewriting, is not aimed at any specific functional language, and it can serve as an intermediate form between functional and machine languages. In general, these intermediate languages serve as a basis for code optimization analysis.

The Intermediate Program Representation(IPR) language is designed to represent the parallelism in the PROOF/L program and analyse it for the efficient exploitation on various parallel computers. The programming language PROOF/L is a parallel object-oriented functional language based on the computation model PROOF. Once a program is written in PROOF/L, the program is transformed to a target code via a two-level translation. To generate a target code for a specific parallel processing system, a PROOF/L code is first translated into an IPR, then the IPR is translated to a target code. The front-end translation makes all the parallelism in the program explicit. Because this step is machine-independent, it needs to be done only once for each PROOF/L program. The back-end translation is machine dependent and the analysis to obtain efficient target codes can be incorporated at this stage. This

two-level transformation approach is shown in Figure 4 1 This chapter is organized as follows: In Section 4 1, the general characteristics of the IPR is presented In Section 4 2, the formal definition of the IPR is given using the Petri net In Section 4 3, the transformation rules from the PROOF/L program to the IPR and from IPR to the target program are presented

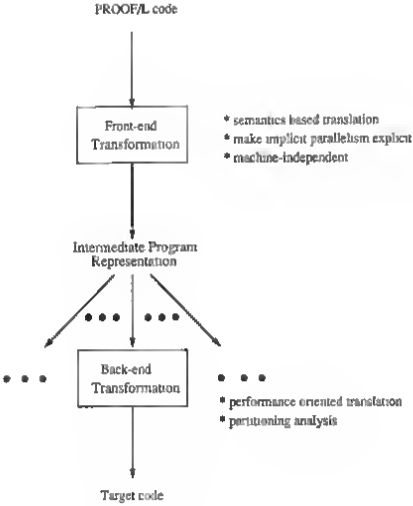


Figure 4-1 Two-level transformation

4.1 Characteristics of the IPR

The IPR consists of two different types of representation one is a Petri net and the other is a set of function nodes and their relations which we will introduce in this section. The semantics for the IPR is also given in the two different levels object level and method level. The object level semantics gives meaning to the object bodies of the PROOF/L program. The method level semantics gives meaning to the function nodes used to represent the methods in the PROOF/L program. This two level semantics makes it easy to understand the important issues in parallel programs, such as communication/synchronization aspects without considering the unnecessary details of the program. This separation of the semantics also allows the verification of programs in different levels, and thus the complexity of the understanding and the verification of programs can be significantly reduced. Before we present the formal definition of the IPR, we summarize the characteristics of the IPR as follows:

- Graphical representation

  A graphical representation can help conceptual understanding of parallel programs on a construct-by-construct basis. A PROOF/L program is represented as a hybrid graph on the two different levels: object-level and method level. At the object level, the IPR is a Petri-net, which is a bipartite directed graph. It is used to represent the method invocation structure of the program. At the method level, the IPR is a directed graph in which each node represents a computation and each directed arc represents precedence relationships between two nodes. The nodes can be divided into two types: computation node and non computation node. The computation node represents a function receiving input value(s) and generating output value(s). These functions are side-effect free, that is, they always produce the same result when the same input values

are given. The non computation node does not represent a specific function, but they are required to exist to represent computation in the IPR. A set of edges between nodes specifies the data dependency between functions.

- Data and control dependency representation

  The precedence relationships among computations at the method level are based on the data dependency among them. A computation associated with a node can be executed when the input data associated with incoming arc(s) is available, and after its execution the result is available for its successor node(s). The functional style of PROOF/L permits to extract data dependency relationships easily among the computations. The dependency relationships among objects are based on control dependency in the sense that the method is not invoked by the availability of data but invoked by the necessity of that method execution. Thus, we can also regard this control-dependency among objects as a demand driven approach. On the other hand, the data dependency among the computations in the method-level can be regarded as a data-driven approach. The effect of such separation of dependency relations needs to be studied more.

- Partitioning analysis

  The IPR can be used to analyze partitioning and allocation of processes to processors. An important factor to minimize the execution time of a program is to determine the proper size of the tasks to be assigned as a sequential code to physical processors. To determine the proper grain size, we need to analyze the tradeoff between communication overhead occurring due to parallel execution of processes. In order to use the IPR for this analysis, we need to annotate the graph with information such as execution time and communication time. The

use of the IPR for grain size analysis and clustering will be explained in the next chapter

4.2 IPR Definition

The IPR is a hybrid graphical program representation in which the high level control structure is specified as a Petri-net and in which the functionality is specified as a data dependency graph The high level control structure corresponds to the bodies of the objects and the low level functionality corresponds to the method definitions within each object The high level structure is called the object-level IPR and the low level representation of functionality is called the method-level IPR.

4.2.1 Object-level IPR

The IPR in the object level is represented as a Petri-net [65] A Petri-net represents the static structure of the system and its dynamic behavior The static structure is represented as a net structure $N = (P, T, F)$, in which P is a set of *places*, representing the availability of data or control, T is a set of *transitions*, representing the activities, and F is a flow relation, representing the dependencies between the places and the transitions To specify the dynamic behavior of the system, the state of the system is represented by a marking in a Petri net, which is a distribution of tokens over places of the net

We define the Petri net to be used as the object-level IPR for the PROOF/L programs in the following

Definition 4.2.1 The Petri net PN is defined as a 5-tuple such that

$$PN = (P, T, F, I, D)$$

in which

P is a set of places,

T *is a set of* transitions,

$F \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs (flow relation),*

I *is a set of initial places, and*

D *is a set of final places*

We also define postsets and presets of places and transitions in the following.

Definition 4.2.2 The preset of a transition t is a set of places such that

$$\bullet t = \{p | p \in P \text{ such that } (p,t) \in F\}$$

Definition 4.2.3 The postset of a transition t is a set of places such that

$$t\bullet = \{p | p \in P \text{ such that } (t,p) \in F\}$$

Definition 4.2.4 The preset of a place p is a set of transitions such that

$$\bullet p = \{t | t \in T \text{ such that } (t,p) \in F\}$$

Definition 4.2.5 The postset of a place p is a set of transitions such that

$$p\bullet = \{t | t \in T \text{ such that } (p,t) \in F\}$$

In $\mathcal{PN}$, there is at least one place p such that $\bullet p = \emptyset$. There also is at least one place p such that $p\bullet = \emptyset$. Let num be a function such that $num(p)$ is number of tokens in a place p.

Definition 4.2.6 The initial place is a place p such that $p \in P$ and $num(p) > 0$ and $\bullet p = \emptyset$.

Definition 4.2.7 The final place is a place p such that $p \in P$ and $p\bullet = \emptyset$.

A marking in the Petri net is changed according to the following transition rules.

1) A transition t is said to be enabled if each input place in $\bullet t$ is marked.

2) A firing of the enabled transition transition t removes a token from each input

place p in st, and adds a token to each output place in t•

When transitions represent computations that are too complex or places that require significant transformation, the transitions or the places can be further refined. Let the refinement of a transition be a sub net sn. The transition can be refined according to the following rules

a) The incoming links and the outgoing links of the transition serve as input and output parameters, respectively

b) There is only one transition that receives all the input parameters

c) There is only one transition that produces all the output parameters.

d) All the transitions except these two transitions and places can interact only with the places and the transitions defined within the sub-net sn

The place can be refined according to the same rules as these except that the transition is replaced with the place

There are two approaches for graphically representing programs: a task interaction graph and a task precedence graph. In the task interaction graph, a program is represented as a undirected graph in which each node represents a task and each edge between two nodes represents a communication relation. Because the edges are non-directional, the execution order relation among the tasks cannot be predicted. The known or estimated communication time can be specified by lumping together all of the communication time required during the program execution between processors.

The task interaction graph is suitable for an analysis of task allocation whose goal is to minimize the total execution time and communication time. In the task precedence graph, a program is represented as a collection of tasks and explicit execution dependencies expressed in the form of precedence relationships. This representation is useful for modeling a program whose behavior can be statically known in advance.

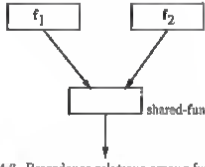


Figure 4 2 Precedence relations among functions

In both graphical representations, the concept of the shared data does not exist, and even simple synchronization cannot be specified For instance, suppose that we have three functions, *f1*, *f2* and *shared-func*, whose relations are shown in Figure 4 2. One interpretation for the relations shown in Figure 4 2 is that the function *shared func* requires both the results of *f1* and *f2* as input in order to execute *shared func*. We call this dependency relation as *value flow* However, there may be situations such that *shared func* may need only one input from either of the two functions Suppose that *f1* and *f2* are clients requesting a service to *shared-func* If *shared func* can serve only one client at a time, the task precedence graph cannot represent this kind of relations among tasks We call this relation as *thread flow*.

In the task interaction graph, such relationship could be implicitly represented. However, due to the lack of a precedence relationship, analysis using precedence relationships cannot be performed on that graph On the other hand, the Petri net can easily distinguish the differences among those cases, as shown in Figure 4.3. The Petri net is chosen due to its capability to distinguish the thread flow and the value flow The problems with the Petri net is that recursive behavior cannot be given the

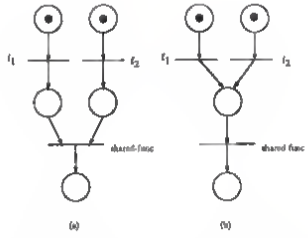


Figure 4.3 The Petri net representation of (a) value flow (b) thread flow

exact semantics. Addition of the inhibitor arc to the Petri net, however, can increase the express power so that the semantics of the recursive behavior can be exactly represented.

4.2.2 Method level IPR

The method-level IPR is a data dependency graph. In the object level IPR each method is represented as a transition. In the method level IPR functionality corresponding to each transition in the object level IPR is represented as a set of nodes and their dependency relations. We present function nodes of the method-level IPR and give the semantics using the Petri net. We define the method level IPR in the following manner:

Definition 4.2.2 The method-level IPR is a directed graph $\mathcal{G}$ such that

$$\mathcal{G} = (V, E)$$

in which

$v_i \in V$ *is a function node, and*

$(v_i, v_j) \in E$ *is an edge representing data dependency from* v_i *to* v_j

The method-level IFR is composed of the following nodes

- A *simple function* node $v_f \in V$ represents a primitive function such as $*, -, +$, ... The input and output for v_f can be represented as follows

  $v_f : I_1, I_2, \ldots, I_m \rightarrow O$ in which $m \geq 1$

- A *constant* node $v_{co} \in V$ represents a constant value generator, which produces the specified same value(s) all the time. There is no input to this v_{co} type of node

  $v_{co} \rightarrow O$ which O is a constant value.

- An *id* node $v_i \in V$ represents an identity function, which always returns input as output

  $v_i : I \rightarrow O$ in which $I = O$

- A *copy* node $v_{cp} \in V$ represents a duplicator, which receives an input and produces the appropriate number of copies having the same value as that input

  $v_{cp} : I \rightarrow O_1, \ldots, O_m$ in which value of I = value of O_i for $1 \leq i \leq m$

- A *macro function* node $v_{mf} \in V$ represents a compound function composed of simple and/or macro functions

  $v_{mf} : I_1, I_2, \ldots, I_m \rightarrow O_i$ for $m \geq 1$

- A *selector* node $v_s \in V$ represents a conditional construction function. v_s receives input data $I_1, I_2, \ldots, I_m$ and control data c and returns an input I_i as an output according to the value of control data c.

  $v_s : I_1, I_2, \ldots, I_m, c \rightarrow O$

- A *distributor* node $v_d \in V$ represents a conditional construction function. v_d receives input data I and control data c and passes I to one of output port O_i according to the value of c.

  $v_d : I, c \rightarrow (O_1, O_2, \ldots, O_m)$ in which $(\ldots)$ means one of the items within $(\ldots)$ is chosen.

- A *merge* node $v_m \in V$ represents a nondeterministic selector, which receives an arbitrary number of input data at a time and returns one, which arrives first. If more than one input arrives at the same time, a single input is chosen arbitrarily.

  $v_m : (I_1, I_2, \ldots, I_m) \rightarrow O$

- A *split* node $v_p \in V$ represents a decomposition function, which decomposes input data and returns a set of data decomposed.

  $v_p : I \rightarrow O_1, O_2, \ldots, O_m$

- A *construct* node $v_c \in V$ represents a composition function, which composes a set of input data and returns them as one element.

  $v_c : I_1, I_2, \ldots, I_m \rightarrow O$

The *selector* and *distributor* nodes were originally introduced in [21], and we generalized them in our method level IPR. The operational semantics for the nodes in the method-level IPR are given using the Petri-net by defining the functionality of the nodes.

The semantics of the basic function nodes, including *constant*, *id*, *macro function* and *copy*, can be represented as a simple net as shown in Figure 4.4. The input places indicate the availability of the input value to the function and the output places indicate the availability of the result of the function application. The semantics of

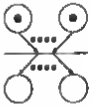


Figure 4 4 The semantics of the function application

the control function nodes, such as *selector*, *distributor* and *merge*, and list handling nodes, such as *construct* and *split*, are represented by the Petri net in Figures 4 5–4 7

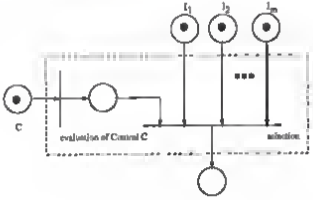


Figure 4 5. The operational semantics of the selector node

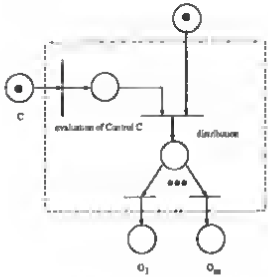


Figure 4 6 The operational semantics of the distributor node

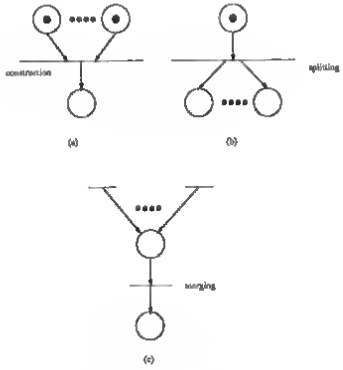


Figure 4.7 The operational semantics of the list handling nodes (a) construct node (b) split node (c) merge node

4.3 Transformation Rules

In this section, we present the transformation rules in the two levels. The PROOF/L program is first transformed into the IPR and then the IPR is transformed to the target code. The former is called front-end transformation and the latter back-end transformation. The front-end transformation is to make all the parallelism explicitly expressed in the IPR. This step is a semantics-oriented transformation and machine dependent issues are not involved at all. The back-end transformation is performance-oriented and the machine dependent parameters, such as communication overhead, number of links and number of processors, are used to perform various analyses. In the following the transformation rules from the PROOF/L program to the IPR and the transformation rules from the IPR to a target code are given. We also show that the transformation from the PROOF/L program to the IPR preserves the correctness of the original PROOF/L program.

4.3.1 Representation of Interactions among Objects

We transform the behavior of objects into an IPR. The behavior of an object is specified in the body of the object. However, the bodies are defined only for active and pseudo-active objects, not for passive objects. The absence of the bodies for the passive objects causes inconvenience for giving semantics for the passive objects. For detailed information regarding these constructs, refer to [82]. To overcome this drawback, we define the interactions among objects using the following constructs.

- SEQuencial execution of methods. When the methods $m_1, m_2, \ldots, m_n$ are executed sequentially in the order $m_1, m_2, \ldots, m_n$, its behavior is specified as SEQ($m_1, m_2, \ldots, m_n$).
- CONcorrect execution of methods. When the methods $m_1, m_2, \ldots, m_n$ are executed concurrently, its behavior is specified as CON($m_1, m_2, \ldots, m_n$).
- WAIT for method invocation. When an object is waiting for the invocation of its

method m by another object O to proceed with its execution, its behavior is specified as WAIT(m, O)

SELect a method for execution based on a condition SEL construct behaves like the CASE statement in ordinary programming languages. When an object selects one of the methods for execution from among the methods $m_1, m_2, \ldots, m_n$ based on a condition, its behavior is specified as SEL$(m_1, m_2, \ldots, m_n)$

– ONE-OF the methods for execution from a group of possible methods: ONE-OF construct is used in cases where different objects could try to invoke the methods defined in the object O simultaneously. The object O permits only one object to invoke its method at a time. This construct serializes the requests and is typically used to describe the behavior of the shared writable objects. Note the difference between the SEL and the ONE-OF construct. Among the set of methods $m_1, \ldots, m_n$ defined in an object, when the object permits only one of its methods to be invoked by other objects, the behavior of the object is specified as ONE $-$ OF$(WAIT(m_1, O_j), \ldots, WAIT(m_n, O_k))$

4.3.2 Front-end Transformation

We present the front-end transformation rules on two levels: object-level and method-level

Object-level transformation

Using the Petri net defined for the object level IPR, we define the transformation rules used in the object level transformation. To do so, we define the compositional semantics of the operators used in expressing the behavior of the objects with the Petri net as follows:

Figure 4 8 Petri net representation of *nil*

Let N_1 and N_2 be the Petri net representations corresponding to the behavior expressions, e_1 and e_2, respectively, defined as follows

$$N_1 = \{P_1, T_1, F_1, I_1, D_1\}$$

$$N_2 = \{P_2, T_2, F_2, I_2, D_2\}$$

in which P_i, T_i, F_i, I_i and $D_i, i = 1, 2$, are disjoint unless specified otherwise.

We first define a special net *nil*

Definition 4.3.1 *nil is a net such that*

$$\mathcal{PN} = (\{p\}, \varnothing, \varnothing, \{p\}, \varnothing)$$

The *nil* net is a net consisting of only one place without any transition No transition is possible in a *nil* net The *nil* net is shown in Figure 4 8

The semantics of the SEQ construct is given in the following definition

Definition 4.3.2 *The transformation rule for SEQ(e_1, e_2) is defined as a net*

$$\mathcal{PN} = \{P_1 \cup P_2, T_1 \cup T_2 \cup t, F_1 \cup F_2 \cup D_1 \times \{t\} \cup \{t\} \times I_2, I_1, D_2\}$$

The SEQ representation requires an addition of a transition t to sequentially connect the two nets The sequential composition of the nets, called *causality*, is shown in Figure 4 8

The transformation rule for the CON construct is given in the following definition

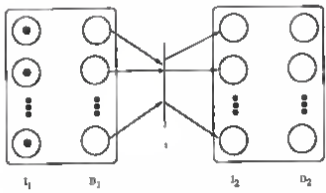


Figure 4.9 The semantics of the sequential invocation

Definition 4.3.3 The transformation rule for CON(e_1, e_2) is defined as a net

$$\mathcal{PN} = \{P_1 \cup P_2 \cup p, T_1 \cup T_2 \cup t, F_1 \cup F_2 \cup (p,t) \cup \{t\} \times (I_1 \cup I_2), \{p\}, D_1 \cup D_2\}$$

The CON representation requires an addition of a place p and a transition t in order to allow the concurrent execution of the two processes represented by each net. The concurrent composition of the two nets is shown in Figure 4.10.

The transformation rule for SEL is given in the following definition.

Definition 4.3.4 The transformation rule for SEL(e_1, e_2) is defined as a net

$$\mathcal{PN} = \{P_1 \cup P_2 \cup p, T_1 \cup T_2 \cup t_1 \cup t_2, F_1 \cup F_2 \cup (p,t_1) \cup (p,t_2) \cup \{t_1\} \times I_1 \cup \{t_2\} \times I_2, \{p\}, D_1 \cup D_2\}$$

The SEL representation requires an addition of a place p and two transitions t_1 and t_2 so that only one transition can be fired each time. The composition of the two nets for selective invocation is given in Figure 4.11.

The transformation rule for ONE-OF is given in the following definition.

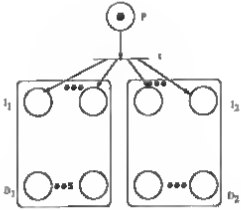


Figure 4.10 The semantics of the concurrent invocation

Definition 4.3.5 The transformation rule for ONE-OF(e_1, e_2) is defined as a net

$$\mathcal{PN} = \{P_1 \cup P_2, T_1 \cup T_2, F_1 \cup F_2, I_1 \cup I_2, D_1 \cup D_2\}$$

in which $| P_1 \cup P_2 | = | P_1 | + | P_2 | - 1$.

Note that there is a common place between the two nets N_1 and N_2. The ONE-OF construct is used to represent mutually exclusive access to the shared object. Such access is specified using the pseudo-function $\mathcal{R}$ in the body of the object. To represent the mutually exclusive access to the shared object, a special place, called the *bottleneck* place, is associated with the transition representing the method invocation which results the modification of the shared object. The bottleneck place for the object is unique, and thus the number of places after the composition of the two nets is reduced by one. This composition is called *fusion of nets via place*, and is shown in Figure 4.12. The transformation rules presented so far show only the composition of the two nets, but these rules can be easily generalized in the case of the composition of more than two nets.

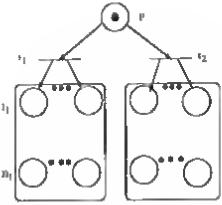


Figure 4 11 The semantics of the selective invocation

In the computation model PROOF, a communication is a point to-point relation in the sense that the communication can only occur between the two objects, the caller and the called The transformation rule for the object communication is given in the following definition

Definition 4 3.6 The transformation rule for the communication between two objects is defined as a net

$$\mathcal{PN} = \{P_1 \cup P_2, T_1 \cup T_2, F_1 \cup F_2, I_1 \cup I_2, D_1 \cup D_2\}$$

in which $| T_1 \cup T_2 | = | T_1 | + | T_2 | - \iota$,

ι *is the number of communication points*

Note in the Definition 4.3 6 that the new net is an union of two nets, but the two sets T_1 and T_2 are not disjoint. If they are disjoint, there is no method in common in both the nets N_1 and N_2 It implies that there is no communication between the two objects The communication between the two objects can only occur when one object invokes a method defined in the other object The method name is known

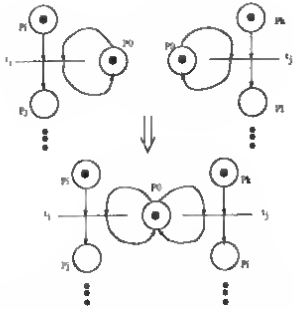


Figure 4 12 The semantics of the mutually exclusive access

in advance to the caller object, and the called object also knows of the existence of the caller object. In the behavior of the caller object, the method name is given with its called object. In the behavior of the called object, the WAIT construct is explicitly used to specify the possible communication. In other words, the existence of communication between the two objects can be detected by examining the Petri-net representation for each body of the two objects. The two nets are composed via the common transition t as shown in Figure 4 13. This composition of nets is called *fusion of nets via transition* and is used only for the representation of communication.

During the transformation, several transitions need to be added to compose the nets. These transitions are not introduced to add functionality to the original nets, but added for the purposes of the composition only. In the following we distinguish

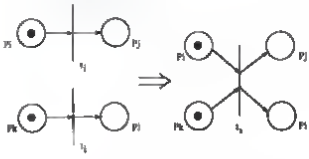


Figure 4.13 The semantics of the communication between two objects

these additional transitions and the transitions corresponding to the functionality of the program

Definition 4.3.7 The useful transition is a transition to represent the method invocation.

Definition 4.3.8 The dummy *transition is a transition used for composition of the nets.*

In the transformation rules, the transitions introduced to compose the original nets are all *dummy* transitions. The firing of the *dummy* transition does not affect the object state

Method-level transformation

We present the transformation rules from the PROOF/L method definition to the method level IPR. In the PROOF computation model, each method consists of an optional *guard* and an expression. Before the expression is executed, the synchronization constraints specified by the *guard* construct should be satisfied. We present the

semantics of the guard mechanism in the Petri net. Because each method is represented as a transition in the object level IPR, the semantics for the guard mechanism can be given by refining a transition using the refinement rules given in the previous section. The guard semantics is shown in Figure 4.14 in which the representation of the transition in the object-level IPR is shown as the two dotted transitions and the boundary of the method is also specified as a dotted rectangle.

The transformation rules for the general function, functional forms and control functions are shown in Figures 4.15 – 4.16. In Figure 4.15, the transformation for a function $f(g_1(x_{11}, x_{1r}), g_2(x_{21}, x_{2r}), \ldots, g_n(x_{n1}, x_{nr}))$ is shown in (a), the transformation for the functional form $\alpha f[z_1, z_2, \ldots, z_n]$ is shown in (b), the transformation for the functional form $\beta[f_1, f_2, \ldots, f_n][z_1, z_2, \ldots, z_n]$ is shown in (c), the transformation for the functional form $\gamma[b_1, b_2, \ldots, b_n][z_1, z_2, \ldots, z_n]$ is shown in (d), the transformation for the functional form $\delta[f_1, f_2, \ldots, f_n][x]$ is shown in (e) and finally the transformation for the functional form $\eta f[x_1, x_2, \ldots, x_n]$ is shown in (f). In Figure 4.16, the transformation for the while $[p\ f\ ][x]$ is shown in (a) and the transformation for the if $[\ p\ f\ g\ ][x]$ is shown in (b).

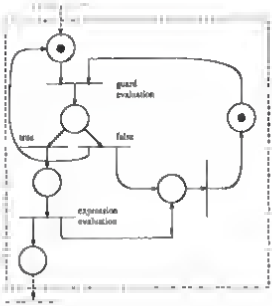


Figure 4.14 The semantics of the method invocation with a guard

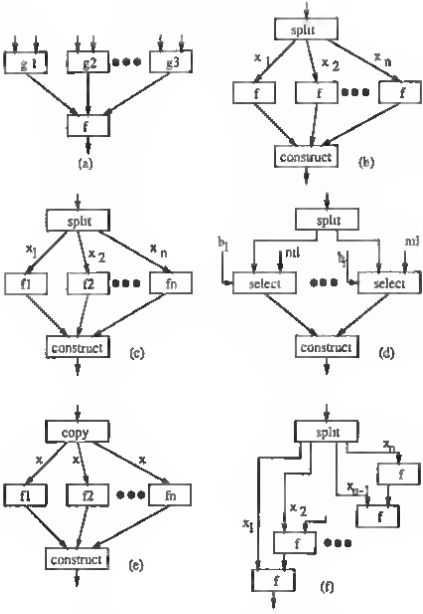


Figure 4.15 The transformation rules for function and functional forms

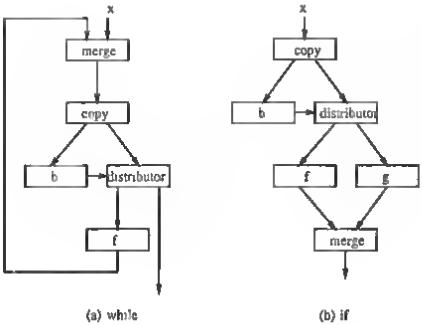


Figure 4.16 The transformation rules for the control functions

4.3.3 Correctness Preserving Transformation

In this section, we show that the front end transformation is a correctness preserving transformation. In order to show the correctness preserving transformation, we give an interleaving semantics to the PROOF/L program by defining a *labeled transition system* and a compositional Petri-net semantics which can be obtained by applying the front end transformation rules to the PROOF/L program. Then we show the interleaving semantics for the PROOF/L program can be retrieved from the Petri net representation of the corresponding IPR form. Note that the Petri net is used to not only give the semantics for the PROOF/L programe but also to give formalism to object level representation in the IPR. In the interleaving semantics, a step of a single process is described as a step of the whole system, changing some global state. Parallel behavior of the system is specified in a sequential interleaved manner. Thus, a computation or program execution is defined as a sequence of such global steps. To give the interleaving semantics for the PROOF/L program we use a *labeled transition system* [51] as an abstract machine.

Definition 4.3.9 *A* labeled transition system *is a structure* $\mathcal{A}$ *such that*

$$\mathcal{A} = (S, \rightarrow, I_0)$$

in which

S *is a set of states,*

$\rightarrow \subseteq S \times Act \times S$ *is a transition relation*

in which Act *is an action to change the state, and*

I_0 *is an initial state*

An element $(s, a, s') \in \rightarrow$ is called a transition (labeled with the action a) from a state s to another state s' by a step a, and will be written as $s \xrightarrow{a} s'$.

In the following, we define a PROOF/L program as a set of processes and then give the semantics to it. Each process corresponds to the body of the non passive object. We can associate a set of local states, called S_i, to each object o_i. Then, the state of the PROOF/L program can be represented as a Cartesian product $S_1 \times S_2 \times \cdots \times S_n$. The actions to change states are represented by constructs, such as `SEQ`, `CON`, `ONE-OF`, `SEL` and `WAIT`. Using the labeled transition system defined in the Definition 4.3.9 and the interpretation of the PROOF/L program given above, the interleaving semantics of the PROOF/ L program can be defined as follows:

Definition 4.3.10 *An interleaving semantics, A_I, for the PROOF/L program, $PROG$, can be represented by the following labeled transition system*

$A_I(PROG) = (S, \rightarrow, I_0)$

in which

S *is a Cartesian product* $S_1 \times S_2 \times \cdots \times S_n$,

I_0 *is an initial state, and*

$\rightarrow \subseteq S \times Act \times S$

in which the transition relations are generated by the following rules

In the rules presented below, $\frac{P}{Q}$ means that if P is true then Q is also true.

Rule 1 Sequential method invocation

$$\mathrm{SEQ}(e_1, e_2) \xrightarrow{a} e_2, \quad a \in Act \text{ corresponding to } e_1$$

Rule 2 Unconditional selective method invocation

$$\frac{e_1 \xrightarrow{a} e_1'}{\mathrm{ONE-OF}(e_1, e_2) \xrightarrow{a} e_1', \mathrm{ONE-OF}(e_2, e_1) \xrightarrow{a} e_1'}$$

Rule 3 Conditional selective method invocation

$$\frac{e_1 \xrightarrow{a} e_1'}{\mathrm{SEL}(e_1, e_2) \xrightarrow{a} e_1', \mathrm{SEL}(e_2, e_1) \xrightarrow{a} e_1'}$$

Rule 4 Parellel method invocation without communication

$$\frac{e_1 \xrightarrow{a} e_1'}{\mathrm{CON}(e_1, e_2) \xrightarrow{a} \mathrm{CON}(e_1', e_2), \mathrm{CON}(e_2, e_1) \xrightarrow{a} \mathrm{CON}(e_2, e_1')}$$

Rule 5 Parellel method invocation with communication

$$\frac{e_1 \xrightarrow{a} e_1', e_2 \xrightarrow{\mathrm{wait}(a)} e_2'}{\mathrm{CON}(e_1, e_2) \xrightarrow{\tau\ \mathrm{wait}(a)} \mathrm{CON}(e_1', e_2')}$$

The set S of PROOF/L terms is defined by the following production system

$$e \ ::= nil \mid \mathrm{SEQ}(e, e) \mid \mathrm{CON}(e, e) \mid \mathrm{ONE-OF}(e, e) \mid e_{a\ \mathrm{wait}(a)}$$

in which $a \in Act$ and $e_{a\ \mathrm{wait}(a)}$ means the communication between two processes via the action labeled a

In the following we prove that the front end transformation preserves the correctness of the PROOF/L program by showing that an interleaving semantics of the PROOF/L program is retrievable from its corresponding IPR.

Theorem 4.3.1 *The transformation rules given by Definitions 4.3.1 – 4.3.6 preserve the meaning of the bodies of the PROOF/L program*

Proof of Theorem 4.3.1 *The proof of this theorem consists of the following two steps*

1) The Petri-net semantics given for the bodies of the PROOF/L program can be

transformed into a labeled transition system.

2) From the labeled transition system obtained in step 1), the interleaving semantics for the original PROOF/L program can be retrieved.

We call a PROOF/L program as $PROG$, and let its IPR representation in the Petri-net be $\mathcal{PN}(PROG) = (P, T, F, I, D)$. We first build a labeled transition system $\mathcal{LTS}$ from $\mathcal{PN}$ as follows.

$\mathcal{LTS}(PROG)$ can be defined as follows:

$$\mathcal{LTS}(PROG) = (S_+(P), \rightarrow, I)$$

where

$S_+(P)$ is a set of non-empty subsets of a set P,

$\rightarrow \subseteq S_+(P) \times Act \times S_+(P)$ is a transition relation, and

Act is a label corresponding to an action and defined as

$s \xrightarrow{a} s', s \in S_+(P)$ $a' \in S_+(P)$ if and only if $\exists t \in T$

$\forall p_1, \forall p_2$ such that $p_1 \times t \in F$ and $t \times p_2 \in F, num(p_1) > 0$

The transition results in $\forall p_1, num(p_2)$ is increased by one.

We show that from $\mathcal{LTS}(PROG)$ an interleaving semantics of the $PROG$ is retrievable by showing that 1) for each useful transition in $\mathcal{PN}$ there is a corresponding action in $\mathcal{LTS}$ and 2) the synchronization among objects in the $PROG$ are not violated.

First, we show that there are only a finite number of dummy transitions introduced during the transformation of any PROOF/L program. For each transformation, there are at most two dummy transitions added. Thus, for a finite number of the bodies in any PROOF/L program, we only need to introduce a finite number of dummy transitions. The dummy and the following useful transitions can be transformed to a new

useful transition without changing the meaning of the original net. Thus, we have proved 1)

Next, from the definition of the transformation rule for communication between two bodies, we know that a transition corresponding to the communication cannot occur unless both the two objects, the caller and the called, are not ready to communicate. Both the caller and called objects can proceed only after the two objects are synchronized for communication (method invocation). We also know by examining the transformation rules given in Definitions 4.3.1 – 4.3.4 that the transformation rules do not violate the method invocation sequences of the original PROOF/L program. Since these transformation rules with the rule for communication are the only rules used for the transformation, the synchronization among the objects cannot be violated. Therefore, we have proved 2)

In conclusion, from 1) and 2), if we give a total ordering to the transitions in $\mathcal{LTS}$, the total ordering corresponds to an interleaving semantics of the original PROOF/L program.

In the following, we show the method level transformation also preserve the correctness of the method definition in the PROOF/L program.

Theorem 4.3.2 The transformation rules for method-level in the IPR preserve the meaning of the methods given in the original definition in the PROOF/L program.

Proof of Theorem 4.3.2 In the PROOF computation model, each method is defined as an applicative function without side-effects within the method. The relations among the statement in each method definition are based on data-dependency and no explicit communication is involved. Thus, it is sufficient to show that the method-level transformation rules do not violate the data dependency relations among functions within the method definition. Since any node in the method-level IPR can be regarded as a

function, by examining the transformation rule for the general function, it is obvious that no violation of data dependency relationship exists Therefore, we show that the meaning the method in the PROOF/L program can be preserved in the representation of the method in the IPR

We can also prove theorem 4 3 2 by building a labeled transition system in which no explicit communication is involved and showing that an interleaving semantics can be retrieved from the semantics of the method level representation of the IPR.

Theorem 4.3.3 The transformation rules we introduced for translation of the PROOF/L program to its corresponding IPR preserve the meaning of the PROOF/L program

Proof of Theorem 4.3.3 From Theorems 4 3 1 and *4 3 2, it is proved*

Lemma 4.3.1 The transformation by the transformation rules given in Definitions 4 3 1 – 4 3 6 preserves the correctness of the original PROOF/L program

Proof of Lemma 4 3 1 By the Theorem 4 3 3, the correctness of the PROOF/L program is preserved in the IPR

4.3.4 Back-end Transformation

Once the IPR is generated from the PROOF/L program, the various analyses are done in the IPR and the modified IPR is generated The analyses includes grain size determination and clustering, which will be explained in the next chapter In this section, the translation rules from the IPR to a given target language are given.

For the body of each object, the IPR is given as a marked Petri net in the front-end transformation In the back end transformation, the marked Petri net for each body can be translated to a local scheduler in each processor A process can be created for each transition in the net and the local scheduler executes the processes in a manner

that the original dependency relationships are not violated In the case of a network of Inmos transputers in which the allocation information of the processors should be known prior to run time, the local scheduling information obtained from the object-level IPR needs to be embedded into the target code Since the net structure can be partitioned, it is not necessary to restrict the boundary of a scheduling unit That is, an object can be partitioned into a set of parallel processes, and a number of objects can be grouped together to form a larger process In either case, the net structure can be partitioned or grouped accordingly

A lock manager needs to be associated to each shared object at this stage. The lock manager controls access to the shared object In the IPR, the existence of the lock manager is implicit in the sense that the complete semantics of the lock manager is hidden During the back-end transformation, a proper concurrency control mechanism can be associated to each shared object

For each primitive function node in the method-level IPR and control functions, we briefly describe the general transformation rules which are not directly targeted to any specific target language To introduce general translation rules, we select basic language constructs available in the existing programming languages These constructs include *assignment statement*(:=), *while loop statement*(**while .. do**), *conditional statement*(**if . then . else**), *case statement*(**switch . case 1 — case n**), *function statement*(**function(input,output)**) and *process creation statement*(**create(process)**)

In the IPR, every node can be regarded as a function which receives input parameters and which returns a value as a result of function application The function can be translated to a simple function or a process according to the following rule.

Rule B1 Function translation

Let *func* be a function. Then *func* is translated either as a single assignment statement or a process creation statement.

rule B1.1: a primitive function

$$O := func(I_1, I_2, \ldots, I_n)$$

ie which if *func* represents primitive operators, such as arithmetic and

logic operators,

$O :=$ operator I_1 when $n = 1$,

$O := I_1$ operator I_2 when $n = 2$

rule B1.2: a user defined function

create$(func(I_1, \ldots, I_n, O))$

The nodes to which the **Rule B1** can be applied include *function*, *id*, *constant*, *copy* and *macro function* nodes.

Control function nodes, such as selector and distributor, are translated according to the following rule.

Rule B2 Control function translation

B2.1: *selector*

when $m = 2$

if c = true **then**

$O = I_1$

else

$O = I_2$

when $m > 2$

switch c

case 1: $O = I_1$

case 2: $O = I_2$

case m: $O = I_m$

B2.2 *distributor*

when $m = 2$

if c= true then

$O_1 = I$

else

$O_2 = I$

when $m > 2$

switch c

case 1: $O_1 = I$

case 2: $O_2 = I$

case m: $O_m = I$

split, *construct* and *merge* nodes are used in the transformation rules for various functional forms as shown in Figure 4.15. In addition to these transformation rules, to utilize the high level programming language constructs, control structures, such as while loop and if then else can also be identified in the IPR.

4.3.5 Examples

Using the transformation rules given in this chapter, various programs have been written in PROOF/L and transformed via the two-step translation to the target programs written in Inmos C for the execution of the Inmos transputer network

The example programs include factorial, producer-consumer problem, dining philosophers problem, warehouse management systems and air-base defense systems. The PROOF/L codes, and the IPR codes for the selected examples are given in the Appendix A2.

CHAPTER 5
ALLOCATION OF PROOF/L PROGRAMS

One of the reasons we execute a program in a parallel processing system is to reduce the time required for completion. It would seem that a simple solution would be to detect all the parallelism in the program and execute it in parallel on the parallel processing system. However, this simple solution does not work in most cases. As the number of parallel processes increases, the communication between processes also increases. Thus, the parallelism needs to be controlled so that the gain of parallel execution cannot be overshadowed by the communication overheads due to excessive parallelism. We identify this controlling parallelism problem as an allocation problem. Various terms used in the literature, such as *allocation*, *partitioning*, *clustering* and *multiprocessor scheduling*, are closely related each other. In general, *task allocation* is used in distributed computing systems, denoting the distribution of tasks into a set of computers. *Partitioning* or *clustering* is to group tasks into a set of partitions or clusters, *multiprocessor scheduling* is to schedule tasks to reduce the completion time. We consider *task allocation* as a general term which also includes the meaning of *clustering*, *partitioning* and *multiprocessor scheduling*. Whenever further distinction is required, we use proper terms based on the definitions. This chapter is organized as follows: In Section 5.1, we present the existing task allocation approaches, discuss their limitations and briefly outline our approach. In Section 5.2, we present a modeling method for representing the PROOF/L program as a directed graph and a simple bottom-up clustering strategy and illustrate it with an example. In Section 5.3, we

illustrate the importance of the grain size determination, present strategies for pipelined parallelism, tree parallelism and graph parallelism, and compare them with the existing approaches.

5.1 Task Allocation Approaches

The problem of task allocation in a program execution in distributed or parallel processing systems has been studied by many researchers, and the existing approaches can be divided into two categories: task allocation without precedence relations and task allocation with precedence relations. Task allocation without precedence relations among tasks can be further divided into three sub-categories: graph theoretic, mathematical programming and heuristic.

In [74], Stone introduced a graph theoretic approach for the task allocation problem in the case of two processors. The program is regarded as a set of tasks, each of which communicates with other tasks. In the program graph, a node represents a task, and an edge between two nodes represents the existence of communication between them. Then the network flow algorithm is applied to this graph. While it is simple to obtain an optimal solution with this method, it can be applied only to a limited number of processors, which severely limits its applicability. An extension of this method for the cases of more than two processors has been proposed in a special case where interprocessor communication patterns are constrained to a tree [75], and a heuristic solution based on this method has also been proposed in [55]. Another method based on a graph has been suggested by Shen in [72]. Shen developed a graph matching algorithm to obtain a task allocation and minimize the completion time of the program when precedence relations among tasks do not exist.

The mathematical programming approach generally involves a 0-1 integer programming technique for assigning programs or data files to processors to minimize

the given cost function with specific constraints [17, 25, 57, 80]. This approach allows constraints to be incorporated into the task allocation model, but is limited by the amounts of the time and space required because the time and the space complexities grow as exponential functions.

While these approaches require exponential time and space to obtain optimal solutions, heuristic approaches can obtain solutions efficiently by sacrificing the optimality of the solution. A popular heuristic approach is the clustering method [24, 37, 68, 69]. In [24], the proposed method consists of two phases. First, a task clustering algorithm is applied to minimize interprocessor communication. Second, to balance the load, underloaded and overloaded processors are identified and adjusted. In [37, 69], a pair of tasks are searched for clustering such that their clustering eliminates the greatest possible interprocessor communication cost. This process continues until all the pairs are clustered. In [68], two heuristic algorithms have been presented. One is an iterative assignment improvement algorithms. Any initial allocation of tasks is transformed by reassigning tasks to obtain a better allocation. The second algorithm is a clustering method.

In the three approaches we have discussed above, the precedence relations among tasks are ignored. When there are precedence relations among tasks, the goal of task allocation is to reduce the completion time. In general, the task allocation problem for minimizing the completion time is known as the multiprocessor scheduling problem. The multiprocessor scheduling problem has been known as an NP-complete problem except in a few very restricted cases [32, 81]. There have been many heuristics proposed to obtain efficient solutions. Among them list scheduling [1] has been popular because of its simplicity and sub-optimality. In list scheduling, the program is represented as a task precedence graph in which each node represents a task and each edge represents the precedence relation between two tasks. Then the list scheduler

assigns a priority to each of the tasks and places tasks in an ordered list according to the priority. When any of the processors is ready to execute, a task with the highest priority is chosen to be executed. Thus, the core of list scheduling is how to determine the priority of each task.

In [50], the path having the longest length from the entry vertex to the exit vertex, called the critical path, is given the highest priority, and each of the remaining tasks is given a priority based on the number of successors to each task. In [67], WP heuristic has been presented. WP heuristic first gives high priority to tasks that compose the critical path. For the rest of the tasks, composite priorities are calculated based on three criteria: tasks having the longest execution time first, tasks having the largest number of successors first, and tasks having largest successor tasks first. Although these heuristics are known to be successful, the limitation is that the communication cost between tasks has been completely ignored. In Kim [52], a heuristic approach, called linear clustering, has been introduced in which not only execution time but also communication cost are considered in scheduling a set of tasks with precedence relations. Hwang[44] introduced a greedy algorithm called ETF(Earliest Task First) in which communication times between tasks are considered. ETF adopts the simple heuristic: the earliest schedulable task is first scheduled to an idle processor. However, these two approaches are completely dependent on the grain sizes determined by programmers and thus the performance of these approaches may not be good enough for some applications in which the grain size is very small.

In addition, these approaches are not directly applicable to allocate the programs based on the computation model PROOF due to the following properties:

- PROOF offers various granularity levels of parallelism: object-level and method level

- The accesses to the shared object need to be mutually exclusive.

To exploit parallelism in different granularity levels naturally implies a multi-level approach in which a different strategy for each level can be used. The problem with such a multi-level approach is that at the highest level[1] it is difficult to obtain appropriate information for the task allocation. In the PROOF programs, for instance, accurate execution time at the object level is very difficult to predict because (a) we do not know how much parallelism can be exploited within the methods, and (b) the execution times of the methods are dependent on input data. Another problem is that the access to the shared data may need to be synchronized. However, in most of the existing allocation approaches, the dependency relationship among tasks is based on data dependency, and thus synchronization requirements cannot be specified.

In order to allocate programs based on the computation model PROOF, we use a two-level allocation approach. At the object level, the parallelism among objects i.e., coarse grain parallelism, is exploited. The objective of object-level allocation is to group the objects into a set of clusters so that communication overhead can be reduced while keeping the potential parallelism among the objects. Once the objects are clustered, each cluster has at most one active object. Thus, it is very likely that in each cluster there is one object busy in execution at the moment. We use as input the object invocation relations which can be obtained from the object decomposition phase. At the method level, the parallelism within each method, i.e., fine grain parallelism, is exploited. At this stage, the proper grain sizes are determined within each method by analyzing the execution and communication times. Depending on the types of the parallelism, different strategies are required to meet the specific requirement. We present grain size determination strategies for the three patterns of parallelism: pipe-lined parallelism, tree-parallelism and graph parallelism. The grain

[1]largest granularity level

size determination strategies can also be used at the object-level if the information about the execution and communication times is explicitly available at this level

5.2 Object Partitioning

The objective of object partitioning is to partition a set of interacting objects so that the communication cost among processors can be reduced while keeping the potential parallelism among them The input for our algorithm is the behavior of the objects in the software system specified using the constructs,such as SEQ, CON, ONE-OF, SEL and WAIT The output of our algorithm will be a set of object clusters and their communication dependency relations The approach consists of the two stages a modeling stage and a clustering stage

5.2.1 Modeling

We begin with modeling the software by a directed graph Each object is represented by one node in the graph, and there is an edge between two nodes if and only if there is a precedence relation between the methods in the two objects Each edge in the graph has a communication weight denoting the degree of communication between the two nodes The communication weight associated with an edge represents the communication cost incurred if the two objects represented by the nodes incident to that edge communicate with one another, but they are not allocated on the same processor.

The software system is modeled by a weighted, directed graph $G = (V, E)$ The graph $G = (V, E)$ has a set of nodes V and a set of edges E such that

- The ith object is represented by a node o_i in V

- An edge (o_i, o_j) is in E if there is a precedence relation between the two objects such that an invocation of a method m_i defined in o_i is followed by an invocation of a method m_j defined in o_j.
- A communication weight w_{ij} is associated to every edge (o_i, o_j).

We assume that the communication weights can be obtained by analyzing the requirement specification. The factors used to determine the weights can include the frequency of the method invocations and the amount of data transfer required for each invocation.

The PROOF/L program is modeled as a directed graph using the following rules. For the simplicity, we use the weight '1' for each possible invocation when we derive the weights in the following rules.

Rule 1. o_1 : CON($o_2, o_3, \ldots, o_n$) describes a case where the objects $o_2, o_3, \ldots, o_{n-1}$, and o_n are executed concurrently after being invoked by the object o_1. It corresponds to a subgraph $G_s = (V_s, E_s)$ where

$$V_s = \{o_1, o_2, \ldots, o_n\},$$

$$E_s = \{(o_1, o_j), \quad 2 \le j \le n\}$$

The communication weights are assigned to the edges as follows:

1. If (o_1, o_j) is new, $w_{1j} = 1$.
2. If (o_1, o_j) is old, new $w_{1j} = w_{1j} + 1$.

Rule 2. o_1 : SEQ($o_2, o_3, \ldots, o_n$) describes a case where o_1 invokes $o_2, o_3, \ldots, o_{n-1}$, and o_n in a sequential order. It corresponds to a subgraph $G_s = (V_s, E_s)$ where

$$V_s = \{o_1, o_2, \ldots, o_n\},$$

$$E_s = \{(o_i, o_{i+1}), 1 \le i \le n-1\}$$

The communication weights are assigned to the edges as follows:

1. If (o_i, o_{i+1}) is new, $w_{ij} = 1$

2. If (o_i, o_{i+1}) is old, new $w_{ij} = w_{ij} + 1$

Rule 3. n_i ONE-OF$(o_2, o_3, \ldots, o_n)$ and o_1 SEL$(o_2, o_3, \ldots, o_n)$ describe a case where o_1 invokes only one $o_j, 2 \leq j \leq n$. Both correspond to a subgraph $G_s = (V_s, E_s)$ where

$$V_s = \{o_1, o_2, \ldots, o_n\},$$
$$E_s = \{(o_1, o_j), \quad 2 \leq j \leq n\}$$

The communication weights in E_s are assigned to the edges as follows:

1. If (o_1, o_j) is new, $w_{1j} = 1/(n-1)$

2. If (o_1, o_j) is old, new $w_{1j} = w_{1j} + 1/(n-1)$

Note that instead of the actual method names the corresponding object names are used in these rules. The above three rules can handle simple clauses in which only one construct is used to specify the behavior of the object. The models using these rules are illustrated in the Figure 5.1.

In the following, we present a rule that can handle nested clauses.

Rule 4. It is applied when nested clauses are used to specify the object behavior. The steps are:

1) Modify the object behavior by substituting all the nested clauses with new objects.

2) Select and apply a rule based on the construct used. For every new object introduced in step 1), do the following steps:

2.1) Apply an appropriate rule and preserve the edge relationships with other objects.

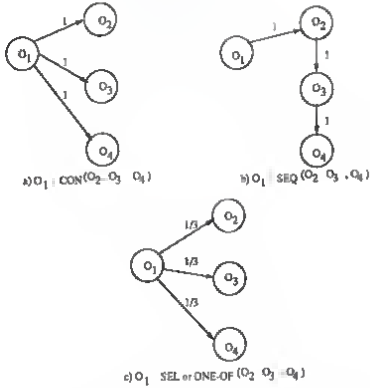


Figure 5 1 The modeling of the object behavior

2 2) Assign communication weights as described earlier in this section.

We now have a graph in which each node represents an object (the same as in the initial graph) and to every edge there is one weight w_{ij} representing the communication costs

5 2 2 Clustering in the Object level

Once the modeling of the software as a directed graph is done, we apply a bottom-up clustering approach to the graph The main objective of this part is to reduce

the unnecessary communication overhead by clustering the objects while keeping the potential parallelism among objects

The input is a weighted, directed graph $G' = (V', E')$ $V' = \{v_1, v_2, \ldots, v_p\}$ where v_i is an object for $1 \leq i \leq p$ Every edge $(v_i, v_j) \in E'$ has a weight w_{ij}. The output is a directed graph in which each node represents a cluster and each edge represents the communication requirements between clusters

The steps we take in clustering the nodes of the graph are as follows.

Algorithm 5.2.1 Object Clustering

input $G' = (V', E')$

output *a set of clusters*

Initially, each object o_i is a cluster $\{o_i\}$.

For each object node a having no incoming edge, do the following

Step 1 *Set a as current object o_a*

Step 2 *Find a node o_b such that (o_a, o_b) is in E' and w_{ab} is the largest*

Step 3 *Cluster $\{o_b\}$ and the cluster which o_a belongs to*

Step 4 *Remove the edges from o_a to the other nodes and from the other nodes to o_a*

Step 5 *If the node o_b has an outgoing edge to any node then*

Set o_b as the current object o_a

Go to Step 2

else

Stop

In Step 1, a node representing an active object is set to a current node to be processed. In Step 2, a node is selected so that the clustering of that node with the current node can reduce the communication overhead most. In Step 3, the node

selected in Step 2 is clustered with the cluster which the current node belongs to In Step 4 the other nodes connected to the current node are disconnected from the current node This step guarantees that all possible parallelism in the software is retained by not clustering any two objects having the possibility of parallel execution. These steps are repeated until all the active objects are processed Because each edge needs to be visited at most once, the time complexity of this clustering approach is $O(e)$ where e is the total number of edges

The result of the object clustering is a set of clusters, each including at most one active object and the objects invoked by that active object Thus, no parallelism among objects has been lost Once the clustering of the objects is complete, each cluster is considered as an independent sub program to be analyzed independently of the other subprograms In each sub-program, each method becomes a subject of analysis In the following, we illustrate object clustering with an example.

To demonstrate the object partitioning, consider the dining philosophers' problem One way to implement this problem is to define a *philosopher* and a *chopstick* as a class Each instance of the *philosopher* class `Philosopher-i` for $1 \leq i \leq 5$, is an active object and each instance of the *chopstick* class, `Stick-i` for $1 \leq i \leq 5$, is a passive object The behavior of the active object `Philosopher-i` can be specified as 'SEQ(think,acquire-stick(i),acquire stick(i+1 mod 5),eat, release-stick(i),release-stick(i+1))' in which 'think' and 'eat' are methods in the `Philosopher`, 'acquire-stick(i)' and 'release-stick(i)' are methods in `Stick-i` and 'acquire-stick(i+1)' and 'release-stick(i+1 mod 5)' are methods in `Stick-i+1` The meaning of each method is self explanatory Since the method invocation defined in the same object is not necessary to specify, the problem can be modeled as shown in Figure 5-2 in which each `Philosopher` is labeled as P_i and each `Stick` is labeled as S_i

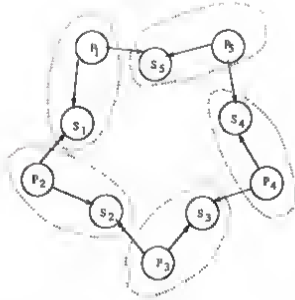


Figure 5.2 An object clustering in the five dining philosopher's problem

When we apply the clustering approach we presented in the previous subsection, a possible solution is shown as dotted lines

5.3 Method Partitioning

As we have discussed above, the parallelism must be managed in such a way that the communication overhead can be controlled The IPR can be used as a task precedence graph for partitioning analysis

In the method partitioning, the proper grain sizes are determined by execution and communication time analysis In the case of cyclic expression, such as looping and recursions, we consider only one pass of such cyclic expression The idea of considering only one pass is to determine the proper grain sizes so that the completion time can be reduced This one-pass approach can be used to determine the optimal

number of processors. Note that the optimal number of processors remains the same regardless of the number of times the expression is repeated. We consider this step the core for exploiting small grain parallelism in comparison to the object partitioning step in which large grain parallelism among objects is exploited. In order to perform the grain size analysis based on the tradeoff between parallel execution and communication overhead, we estimate the execution time of each node in the IPR and the communication time between the two adjacent nodes by analyzing the assembly code corresponding to the target code for each IPR construct. In the following sections, we present the existing grain size analysis techniques, and our grain size determination approaches on three different types of parallelism: pipelined parallelism, tree parallelism and graph parallelism.

5.3.1 Grain Size Analysis

In this section, we briefly compare two different strategies to determination of grain sizes and present our methods on various types of parallelism.

The existing grain size determination strategies can be divided into two categories based on how the grain size is determined: programmer control and automatic determination. In the programmer controlled approach, programmers are fully responsible for determining the grain sizes, as well as expressing explicitly parallelism. The programmers can use parallel language constructs indicating the tasks to be executed in parallel. When a programmer has specific information about the behavior of a program, the programmer can determine the sizes of tasks. When that program is ported to a different parallel computer, the sizes of tasks need to be changed to best fit to the new processors. In addition, it may not be easy for the programmers to make decisions on the sizes of the tasks due to lack of information. This approach

is closely related to the second category of parallel programming approach in which parallel language constructs are used for expressing parallelism and communication.

On the other hand, in the automatic determination approach, grain sizes are determined automatically. This approach can be further divided into two classes: the compiler approach and run time approach. In the automatic determination at compile time approach, the programmers do not provide any information regarding the granularity. During compilation, heuristics are used to statically determine the sizes of tasks [58, 59, 70]. One disadvantage is that some information may not be available before the run-time.

In the run time approach, only simple heuristics can be applied to determine the sizes due to the costly overhead. For example, as in [34], each recursive function call creates a new task to be assigned to a processor. In this case, since functional programming involves frequent recursive function calls, it is likely that too many small tasks will saturate the system. In general, such run time approaches ignore the size of tasks under the assumption that there are reasonably many processors available. The automatic determination approach looks more promising since the programmers need not worry about the grain size at all. However, some parallelism may not be detected without help from programmers.

In our approach, we integrate the two strategies by utilizing information available at the compilation time and also allowing programmers' control. The grain size is determined based on the analysis of the execution and communication times which can be obtained during the compilation time. The programmers can analyze the IPR and select proper strategies depending on the characteristics of the parallelism.

We make the following assumptions about the underlying MIMD parallel computers.

- The computer system consists of fully connected identical processors having the same processing capability
- Each processor has a capability of performing program execution and I/O simultaneously
- The communication cost between two processors depends only on the data size to be transmitted. Currently, we ignore the time required to set up the communication
- Communication cost between two tasks residing on the same processor is small enough to ignore

We define the notations we use in this section as the following

Definition 5.3.1 An execution time for a node n, denoted as e(n), *is a time required to complete the computation represented by n without being interrupted.*

Definition 5.3.2 A communication time between two nodes m and n, denoted as c(m,n) *is a time required to transmit data from m to n under the assumption that m and n are assigned to the adjacent processors*

Definition 5.3.3 A completion time for a code c with u number of processors, denoted as E(c,u), *is a time required to finish all the computations and communication involved in the code c*

We illustrate the importance of grain size in parallel processing with an example. Suppose that we have a PROOF/L code, called M1, such as e(b (d(v_1), e(v_2)), c (f(v_3), g(v_4))). The IFR for M1 is shown in Figure 5.3. The graph consists of seven nodes, e, b, c, d, m, f, g, and six edges, which represent data precedence

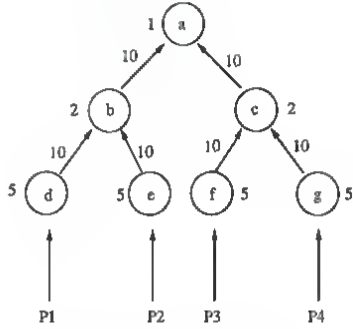


Figure 5 3 An IPR representation for M1

relations. This tree-type parallelism is a typical form resulting from a divide-and-conquer algorithm

Assuming that there are enough processors, a simple approach would be to assign four tasks d, e, f and g to four different processors P_1, P_2, P_3 and P_4, respectively After completing the execution of these four tasks, P_1 and P_2 continue the execution of the nodes b and c, respectively Then, P_1 executes a to complete the execution In this case E(M1,4) can be calculated as follows.

$$E(M1,4) = 5 + 10 + 2 + 10 + 1 = 28$$

This result is not desirable, since if we only utilize one processor, $E(M1,1) = \sum_{x=a}^{g} e(x) = 5 \times 4 + 2 \times 2 + 1 = 25$ Thus $E(M1,1) < E(M1,4)$, and there is no gain in parallel processing because the communication overhead has overshadowed the gain of parallel execution

5.3.2 Pipelined Parallelism

One of the patterns of parallelism is pipelined parallelism. In exploiting pipelined parallelism, one important consideration is the number of the segments, that is, how to divide the entire process into a set of segments to reduce the completion time. In the following, we show that we can find the optimal size of segments, i.e. the proper grain sizes.

Definition 5.3.4 *A dominant segment is a sub process in a pipelined process such that the completion time of the entire process is dominated by that sub-process.*

Definition 5.3.5 *All the sub processes other than the dominant segment are called* subordinate segments.

The dominant segment dictates the completion time of the entire pipelined process. In other words, the dominant segment is always busy once all the segments are filled with data.

Definition 5.3.6 *A* computation *segment is a sub process whose task is to receive input data, execute the computation with it and return the result.*

Definition 5.3.7 *A* communication *segment is a sub process whose task is to pass data from the preceding computation segment and to the succeeding computation segment.*

Definition 5.3.8 *A* one-pass completion time, $E(s_i)$ *defined for a segment* s_i, *is an amount of time required to complete processing of a datum in* s_i.

Note that the dominant segment can be either a computation segment or a communication segment. Computation segments and communication segments have different properties. Computation segments can be divided to reduce the one-pass time.

However, communication segments cannot be so refined. This distinction implies that in order to find the optimal grain sizes we need to begin the analysis from the smallest grain size available. In our approach, the IPR presents the smallest grain parallelism we can get in the program statement level.

We make the following assumptions before we introduce an optimal solution for the determination of the proper grain sizes.

- The one-pass completion time for each segment is fixed and known a priori.
- There are a sufficiently large number of data elements.
- The execution of code and communication can be done simultaneously.
- The communication link can deliver only one result at a time.

Lemma 5.3.1 There is always a dominant segment in a pipelined process.

Proof of Lemma 5.3.1 Under the assumption that each segment requires a fixed amount of time for completing its processing of a datum, the proof is trivial.

The goal of finding proper grain sizes in a pipelined process is to reduce the one-pass completion time of the dominant segment either combining a dominant segment and its neighboring segments or refining a dominant segment. However, since the IPR presents as fine parallelism as possible, the refinement of the dominant segment would not be considered.

Algorithm 5.3.1 Determine Grains for Pipe-based Parallelism
input *An IPR representing pipelined parallelism*
output *Optimal grain sizes*

Step 1 *Sort the segments using one pass completion time as a key in descending order*

Step 2 *Find a dominant segment Let the dominant segment be S_i and its preceding and succeeding segments be S_{i-1} and S_{i+1}, respectively*

Step 3 **If** *S_i is a communication segment* **then**

If *$E(S_{i-1}) + E(S_{i+1}) < E(S_i)$* **then**

Delete S_{i-1}, S_i and S_{i+1} from the list

Add new S_i into the list such that

$E(S_i) = E(S_{i-1}) + E(S_{i+1})$

Go to Step 2

else

Stop

In the following it is shown that the algorithm 5 3 1 yields an optimal solution

Theorem 5.3.1 The algorithm 5 3 1 always finds optimal grain sizes.

Proof of Theorem 5 3 1 By Lemma 5 3 1, the Step 2 will always find a dominant segment When the dominant segment is a computation segment, since we cannot reduce the computation any further, the grains cannot be further clustered Thus, the dominant segment remains the same, and so does the completion time of the entire process Now consider a case in which the dominant segment is a communication segment In this case the algorithm tries to partition the dominant segment with its neighbor computation segments to reduce the one-time pass time Let the sequence of the partitioning obtained by the algorithm 5 3 1 be called P1 Assume that there is another sequence of the partitioning called P2 which can lead to a better solution than

P1 *In P2, a communication segment S_i is first partitioned with its neighbors S_{i+1} and S_{i-1} before S_i when*

$$E(S_i) > E(S_j) \tag{1}$$

Partitioning of S_i with its neighbors S_{i+1} and S_{i-1} means that

$$E(S_{i-1}) + E(S_{i+1}) < E(S_i) \tag{2}$$

From (1) and (2) we know that $E(S_{i-1}) + E(S_{i+1}) < E(S_i)$ (3)

If S_i and S_j are two adjacent communication segments, then in P2 it may not be possible to partition S_i with S_{i-1} and S_{i+1} ($= S_{j-1}$) because $E(S_{i+1})$ has increased to $E(S_{i+1}) + E(S_{i+1})$. Thus, the dominant segment S_{i+1} may remain unpartitioned in P2. Therefore, P2 cannot yield an optimal solution. We proved by contradiction that the algorithm 5.3.1 can always yield an optimal solution.

Let the number of segments be n. Then, the algorithm 5.3.1 can find an optimal grain sizes during compilation in the time complexity shown in the following theorem.

Theorem 5.3.2: The time complexity of the algorithm 5.3.1 is $O(n\log n)$.

Proof of Theorem 5.3.2: Step 1 requires $n\log n$ steps. By using a priority queue data structure to store the sorted list, we can retrieve and store any element in $\log n$ step. Thus, each execution of Step 3 requires at most $4\log n$ steps for three deletions and one addition to the priority queue. Since there are at most n segments, the entire algorithm can run $O(n\log n)$.

5.3.3 Tree Parallelism

We consider a case in which the task precedence graph is a tree. Before we present our approach, we define the terms we use.

Definition 5.3.3: The one-level sub-tree is a subset of nodes $v_0, v_1, \ldots, v_n$ in a tree such that v_0 is a parent node of all the nodes $v_1, v_2, \ldots, v_n$.

In the following, we call the one-level sub-tree simply sub-tree. The number of sub-trees in a tree is the same as the number of non-leaf nodes.

Definition 5.3.10 *A task precedence tree T_p is a tree in which each node represents a computation and each edge specifies the data dependency relationships among nodes.*

Parallelism obtained from divide-conquer strategy can lead to the parallelism of tree pattern and thus be represented by T_p.

Definition 5.3.11 *A gain-tree T_g of T_p is a weighted tree in which each node, called a gain node, represents a sub tree in T_p and each edge represents data dependency relations among the nodes. Each gain node has a weight, called gain, corresponding to an amount of possible maximum contribution to reducing the completion time when the corresponding sub tree is clustered.*

A gain for a sub-tree consisting of $n_1, \ldots, n_m$ is denoted as $GAIN(n_1, \ldots, n_m)$. Our grain size determination approach can be considered as a *horizontal* clustering or partitioning in that a set of adjacent nodes, i.e. a sub-tree, is considered as a candidate for clustering. The essential part of our grain size determination approach is to estimate the possible contributions which can be made by clustering the adjacent nodes. Our approach consists of two parts:

1) build a gain tree from a given input task precedence graph, and

2) determine grain sizes from the gain tree.

The gain tree can be built by analyzing each sub-tree in the task precedence tree using the following procedure.

Procedure Gain Analysis

input: a sub-tree, consisting of s and its children $n_1, \ldots, n_m$

output: $GAIN(s, n_1, \ldots, n_m)$

Step 1 Calculate the total execution time tc in one processor

$$tc = c(s) + \sum_{i=1}^{m} c(n_i)$$

Step 2 Find a node n_k, for $1 \leq k \leq m$, such that

$$pc = c(n_k) + c(n_k, s) \text{ is the second largest}$$

Step 3 If $pc + c(s) > tc$ then

$$GAIN(s, n_1, n_2, \ldots, n_m) = pc + c(s) - tc$$

else

$$GAIN(s, n_1, n_2, \ldots, n_m) = 0$$

In Step 1, the bottleneck time used to decide whether the computation represented by a sub-tree is or is not executed in parallel is determined by summing all the execution times of the nodes in the sub tree. In Step 2, the time required to complete the computation represented by the sub-tree when the nodes in that sub-tree are not clustered is calculated. Because at the scheduling stage one of the child nodes can be scheduled to the same processor as the node s and by the static analysis the scheduler can choose a node n_i such that $c(n_i) + c(n_i, s)$ is the largest, the second largest schedule length is calculated to be used as an actual time required to complete the computation represented by $s, n_1, \ldots, n_m$. In Step 3, a gain is calculated by comparing the bottleneck time with the actual completion time calculated in Step 2. If there is a positive gain, then the amount of the gain calculated is assigned to the gain node. Otherwise the gain is set to zero.

The time complexity of the procedure Gain Analysis can be determined by the following analysis. Step 1 requires $O(1)$ time, Step 2 requires $O(m)$ time in which m is a number of child nodes and Step 3 requires $O(1)$ time. Thus, the exact time complexity of the procedure Gain Analysis is $O(m)$.

Now, we build a *gain tree* T_g from a precedence tree T_p using the procedure Gain Analysis in the following manner:

Algorithm 5.3.2 Build Gain Tree

input *a task precedence tree* T_p

output *a gain tree* T_g

For all sub-trees t_s *do*

Let t_s *consist of a node* s *and its children* $n_1, n_2, \cdots, n_m$

Step 1 *Call* `Gain-Analysis`$(s, n_1, n_2, \cdots, n_m)$

Step 2 *Connect the gain node to the existing gain nodes*

In Step 1, Gain Analysis is called for each sub-tree to determine the possible contribution to the reduction of the completion time when the sub-tree is clustered. In Step 2, the newly created gain node for the current sub tree is connected to the existing gain nodes. The construction of the gain tree can be done by omitting leaf nodes from the original task precedence tree and associating the gain calculated in Step 1 to a parent of the corresponding sub-tree. Step 1 requires $O(m)$ where m is the number of the child nodes. Since Step 1 is executed for each sub-tree and the number of sub-trees is bounded by the number of non leaf nodes in the tree, the algorithm needs to visit each node once. In Step 2, each node also needs to be visited once. Thus, the time complexity of the algorithm 5.3.2 is $O(n)$ where n is the number of nodes in the tree.

Once the gain tree is built, the grain size can be determined by selecting clusters heuristically. Our grain size determination is based on the observation that contribution from the nodes close to the root node[2] propagates to the other nodes. In order

[2]the root node in a tree is a node having depth of 0

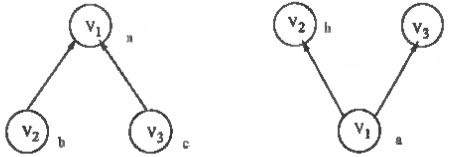


Figure 5.4 Simple gain tree examples

to illustrate this, suppose that we have simple gain trees as shown in Figure 5.4 in which $v_i, 1 \leq i \leq 3$, represent a set of gain nodes and a, b, c represent an amount of the gain for each node. In Figure 5.4 (a), the precedence relations are $v_2 \rightarrow v_1$ and $v_3 \rightarrow v_1$. In Figure 5.4 (b), the precedence relations are $v_1 \rightarrow v_2$ and $v_1 \rightarrow v_3$. The goal of the analysis is to select the best two candidates for clustering so as to increase the overall contribution to the reduction of the completion time. The overall contribution is determined by the following equation

overall contribution
= min(overall contribution from a path between v_1 and v_2,
overall contribution from a path between v_1 and v_3)

Thus, when $\min(b, c) \leq a$, v_1 and a node having a weight of $\max(b, c)$ are chosen to cluster, and when $\min(b, c) > a$, v_2 and v_3 are chosen.

Note that the rules presented above can be applied to both gain trees shown in Figure 5.4. It implies that our gain size analysis technique can be used for the analysis

of both in tree form as in Figure 5.4 (a) and out tree form as in Figure 5.4 (b). The following is an algorithm to determine grain size:

Algorithm 5.3.3 Determine Grain Size

Input *A gantt tree T_g*

output *A clustered task precedence tree*

Step 1 *Initialize all the nodes as 'unclustered'*

Step 2 *Sort the nodes in T_g using the gain as a primary key in descending order and the depth of each node as a secondary key in ascending order*

Step 3 *Get the largest node n_l in the sorted list*

Step 3.1 *If n_l is not a root node and a parent of n_l and any sibling of n_l are 'clustered' then go to Step 3.4*

Step 3.2 *If two or more child nodes of n_l are set 'clustered' then go to Step 3.4*

Step 3.3 *Set n_l as 'clustered'*

Step 3.4 *Delete n_l from the list*

Step 3.5 *If there is a node with a positive gain then go to Step 3 else Stop*

The algorithm 5.3.3 determines which sub-trees need to be clustered by analyzing gains of the possible candidates. Step 1 requires $O(n)$ time to visit each node once. In Step 2, $O(n\log n)$ time is required to sort n number of nodes. Step 3 must visit

adjacent nodes at most once for each node. Because the number of the adjacent nodes is same as the number of edges in a node and each edge will be visited at most twice, the overall time complexity in Step 3 is bound to $O(e)$ in which e is the number of the edges in T_p. Therefore the time complexity of this algorithm is $O(max(nlogn, e))$. Note that in the case of trees the complexity yields to $O(nlogn)$. Since this algorithm is also used in the case of the graphs later, we define the time complexity of this algorithm as above.

In the following, we compare our approach to McCreary's approach [58]. McCreary uses an algorithm of the complexity $O(n^3)$ that decomposes a graph into a set of *clans* that are classified as primitive, linear, or independent. When the clans are labeled as independent, the possibility of parallelization exists. However, when the clans are labeled as primitive or linear, they are grouped together and executed sequentially. In order to compare our approach to McCreary's, we use the same example used in [58]. The task precedence tree of the example is shown in Figure 5.5. Every node has a unique number within the circle representing that node, and a weight is attached to the node. The communication cost is assigned to each edge. For instance, a node 9 has a weight 1 and each of the three edges has communication cost 18. Using McCreary's approach, the schedule and its completion time are shown in Figure 5.6.

The problem with McCreary's approach is that it begins to search the candidates for parallel execution from the bottom of the tree. Once the clustering is done at the lower part of the tree, the clustering at the higher level is less likely to occur, since such clustering may have to sacrifice the possibility of parallel execution at the lower level. In addition, the key concept of the graph decomposition approach, *clan*, is determined without using information regarding the execution time and communication cost.

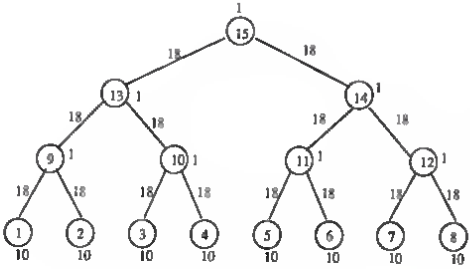


Figure 5.5 A task precedence tree

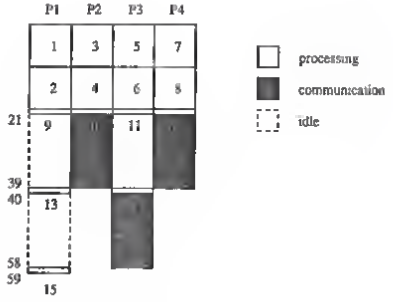


Figure 5 6 A schedule obtained from McCreary approach

Figure 5.7 A gain tree for the tree parallelism example

Our grain size determination approach uses the execution and communication times to determine the proper grains at the beginning. By analyzing the contribution of the clustering locally in each sub tree, we build the gain tree. Then, we first select the largest gain in the gain tree as the candidate for the clustering and continue the selection until all the positive gains are processed. The gain tree for this example is shown in Figure 5.7 in which a calculated gain is shown within each gain node. The corresponding nodes in the task precedence tree shown in Figure 5.5 are also associated with each gain node.

From the information obtained in the gain tree, we can determine five grains as follows:

C1 = { 1, 2, 9, 10, 13, 14, 15 }

C2 = { 5, 6, 11 }

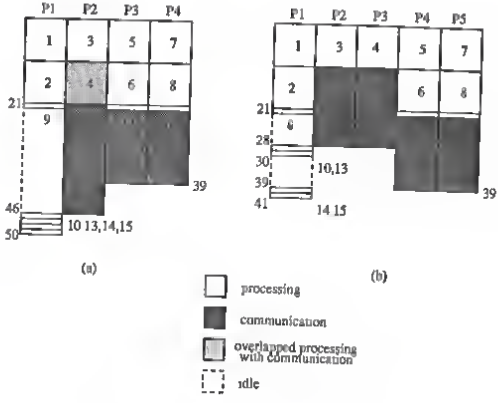


Figure 5.8 A schedule obtained from our approach

C3 = { 7, 8, 12 }

C4 = { 3 }

C5 = { 4 }

Using these clusters, we show two schedules and their completion times in Figure 5.8. A schedule for four processors is shown (a) in Figure 5.8 and a schedule for five processors is shown (b) in Figure 5.8

In the following, we present a task allocation strategy which can produce an optimal solution in a restricted case. We first present the algorithm and then show that it can find an optimal solution when the following assumptions are met

- The task precedence graph is a tree
- There are sufficiently many processors so that the leaf nodes in the task precedence graph can be executed simultaneously, if necessary
- The grain size analysis has been completed and no more clustering of grains to reduce communication time is necessary

This algorithm can be considered as a variant of list scheduling. The priority of the task for allocation to the same processor is decided by the urgentness of the task. This algorithm also uses the concept of the gain analysis. The algorithm tries to reduce the completion time of the each node by clustering the critical path with that node

Algorithm 5.9.4 Task allocation for tree

input *A task precedence tree*

output *a set of clusters*

Step 1 *Identify a node s whose predecessors are all leaf nodes, n_i, for $1 \leq i \leq m$. If there is no such a node s, then stop*

Step 2 *If s has only one predecessor, n, then replace s, n and an edge between s and n with a new node n' such that $e(n') = e(s) + e(n)$ and go to step 1*

Step 3 *For s and n_i, $1 \leq i \leq m$, do the following*

Step 3.1 *Calculate the total execution time in one processor*

$$tc = e(s) + \sum_{i=1}^{m} e(n_i)$$

Step 3.2 *Find a node* n_p *for* $1 \leq p \leq m$, *such that*

$$e(n_p) + c(n_p, s) \text{ is the largest.}$$

Step 3.3 *Find a node* n_q, *for* $1 \leq q \leq m$, *and* $p \neq q$, *such that*

$$c_2 = e(n_q) + c(n_q, s) \text{ is the largest}$$

Step 3.4 *If* $c_2 > tc - e(s)$ *then*

cluster s *and* n_i *into a new node* y *such that*

$$y = s \cup n_1 \cup n_2 \cdots \cup n_i \text{ and } e(y) = tc$$

delete s *and all* n_i *from the tree*

attach y *in the place of* s

else

make a cluster for each n_i *for* $1 \leq i \leq m$, $i \neq p$

cluster s *and* n_p *and represent it as a new node* r

such that $r = s \cup n_p$ *and*

$$e(r) = max(e(n_p) + e(s), e(n_q) + c(n_q, s) + e(s))$$

delete s *and all* n_i *from the tree*

attach r *in the place of* s

Step 3.5 *Go to step 1*

We show in the following theorem that Algorithm 5.3.4 produces an optimal allocation in the restricted case

Theorem 5.3.3 Algorithm 5.3.4 produces an optimal allocation when the following assumptions are satisfied

1) The task precedence graph is a tree 2) There are a greater number of processors

than the number of leaf nodes in the task precedence graph. 3) No more clustering is required in the sense that the grains of proper size have been obtained.

Proof of Theorem 5.3.3: If assumption 3) is satisfied, c_2 is always less than or equal to $tc - e(s)$ in step 3.4 of Algorithm 5.3.4. Thus, the else part of step 3.4 will always be executed. Assumptions 1) and 2) imply that the finishing time of the node s (the time required to complete the node s and its descendants) is only dependent on how the node s and its descendants are allocated. Thus, we only need to show that Algorithm 5.3.4 generates an optimal solution for a subgraph of the given task precedence graph. Suppose that there is a subgraph consisting of a node s and its predecessors n_i, for $1 \leq i \leq m$. Let the allocation produced by Algorithm 5.3.4 be A, and the time required to complete the tasks in this subgraph using A be C_A. Suppose that we have another allocation, called B, for this subgraph, and the time required to complete it in B is C_B, and $C_B < C_A$. In the following, we show that $C_B < C_A$ cannot be true. In A, all n_i except n_p will be allocated to different processors, and in each cycle of Algorithm 5.3.4 only one task is chosen to cluster with its successor, and the addition of any other tasks to this cluster will not reduce the completion time by the assumption 3). In B a different task, $n_{p'}$, other than n_p, would be selected in step 3.2 to cluster with its successor s. In step 3.3, n_p will be chosen instead of n_{q}, because $e(n_p) + c(n_p, s)$ is the largest.

Then, $C_B = max(e(n_p) + c(n_p, s) + e(s), e(n_{p'}) + e(s))$

Since we know that $e(n_p) + c(n_p, s) > e(n_{p'})$,

$$C_B = e(n_p) + c(n_p, s) + e(s) \quad (1)$$

In A, $C_A = max(e(n_p) + e(s), e(n_q) + c(n_q, s) + e(s))$ (2)

Therefore, there are two cases.

case 1) if $e(n_p) + e(s) \geq e(n_q) + c(n_q, s) + e(s)$ then in (2)

$$C_A = e(n_p) + e(s) \tag{3}$$

From (1) and (3), we know that $C_B \geq C_A$ is a contradiction

case 2) if $e(n_q) + c(n_q, s) + e(s) > e(n_p) + s(s)$ then in (2)

$$C_A = e(n_p) + c(n_q, s) + s(s) \tag{4}$$

From (1) and (4), to satisfy $C_B < C_A$, the following equation should be true.

$$e(n_p) + c(n_p, s) + e(s) < e(n_q) + c(n_q, s) + e(s)$$

This contradicts the assumption that $e(n_p) + c(n_p, s)$ is the largest Thus, no such allocation B exists

We show the complexity of the Algorithm 5 3 4 in the following theorem

Theorem 5.3.4 The Algorithm 5 3 4 can run in the time complexity of $O(n)$ when n is the number of nodes in a given IFR

Proof of Theorem 5.3.4 Step 1 can run $O(1)$ time Step 2 can also run in $O(1)$ time Step 3 1 requires $O(1)$ time Steps 3 2 and 3 3 require visiting i number of nodes, in which i is the number of predecessors, and thus $O(i)$ time is required. Step 3 4 also requires visiting i number of nodes, and thus $O(i)$ time is needed to complete the step Once they are visited, each node will not be visited again since they are clustered at this step Thus, the entire algorithm requires visiting at most once each node Therefore the time complexity of this algorithm is $O(n)$

5.3.4 Graph Parallelism

Now we extend our grain size determination approach to the general cases in which the task precedence relations are represented by a directed graph We first define *depth* and *height* in the directed graph

Definition 5.3.12 A depth of a node in a graph is the length of the longest path from the highest ancestor of that node

A node having no incoming edge is of depth 0

Definition 5.3.13 A height of a graph is the largest depth of the graph

A *gain graph* G_g and a *task precedence graph* G_p are defined in the same manner as the *gain tree* and *task precedence tree* except that they are graphs. The nodes in the gain graph are called gain nodes to distinguish them from the nodes in the task precedence graph. In the graph cases, we also analyze gains to determine the proper grain sizes. We first build a gain graph and apply the algorithm 5.3.3 to determine grain sizes. The following is an algorithm to build the gain graph.

Algorithm 5.3.5 Build Gain Graph

input *A task precedence graph* $G_p = (V, E)$, $|V| = n$ and $|E| = e$

output *A gain graph in which each node represents a sub-tree in the graph*

Step 1 *Determine the depth of each node in* G_p

Step 2 *Set all the nodes 'unmarked'*

Step 3 *For a depth* $d = 0$ *to a height* k *do*

For each node s *of depth* d *do*

Step 3.1 *Find predecessors,* $n_1, n_2, \ldots, n_m$ *of* s *having depth* $d-1$

if they are not all 'marked' then

Call Gain Analysis$(s, n_1, n_2, \ldots, n_m)$

Set $n_1, n_2, \ldots, n_m$ *'marked'*

if the out degree of s *is greater than 1 then*

set s *'marked'*

Connect the gain node with the existing gain nodes

Step 3 *2Find the successors* $n_1, n_2, \ldots, n_m$ *of* s *having depth* $d+1$
Call Gain-Analysis$(s, n_1, n_2, \ldots, n_m)$
Set $n_1, n_2, \ldots, n_m$ *'marked'*
Connect the gain node with the existing gain nodes

In Step 1, the depth of each node can be determined using a breadth first search method. Each node may be involved in more than one path, and consequently can have a different length for each path. In such cases, by the definition of *depth*, the largest value is set to the depth of that node. In Step 2, all the nodes are initially set as 'unmarked', meaning that the node is not involved in any clustering. In Step 3, the gain graph is built by visiting each node. First, a node having smallest depth is chosen, and its predecessors are first checked to determine whether all of them are already included in any cluster. If they were not included in any cluster, then *GAIN* is calculated. The predecessors are all set 'marked', and the node is set 'marked' unless it has only one successor. A gain node representing the nodes under consideration is created. Edges are established from the existing gain nodes to the newly created gain node if there is a node common to both the new gain node and the existing nodes. The same steps are applied to the successor nodes except that all the nodes set 'marked'.

The time complexity of this algorithm is analyzed as follows.

Steps 1 and 2 require visiting each node once, and thus $O(n)$ time is required. The complexity of Step 3 is dominated by the procedure Gain-Analysis, and thus each pass of Step 3 requires $O(m+m')$ where m and m' are the number of predecessors and the number of successors, respectively. Step 3 needs to be executed for each node in the graph. Because $m+m'$ is the total number of edges in a node, each edge

needs to be visited at most twice and thus the time complexity of this step is bound to $O(e)$. Therefore, the overall complexity of the algorithm 5.3.5 is bound to $O(e)$.

Once the gain graph is built, we can apply the algorithm 5.3.3 to determine the proper grains. Then the newly-obtained task precedence graph as a result of applying the algorithm 5.3.3 can be used as input for the scheduling stage.

To illustrate our approach we use the example used in [59]. The task precedence graph for the FFT (Fast Fourier Transformation) is shown in Figure 5.9 using the same notation as the former example. Using the algorithm 5.3.5, the gain graph is obtained as shown in Figure 5.10.

We also show a schedule for the FFT problem in the case of four processors in Figure 5.11. McCreary [59] extended her approach by adding analyzing techniques for some special patterns of dependency relations among primitive nodes. Note that with her improved method the former task precedence tree example shown in Figure 5.5 results in the same schedule because the former example does not include primitive nodes. Compared to McCreary's approach, our approach has the following advantages. First, our approach is more efficient in terms of time complexity. As shown above, the time complexity of our approach is $O(max(nlogn, e))$ in which $nlogn$ is for the algorithm 5.3.3 and e is for the algorithm 5.3.5. The time complexity of McCreary's approach is dominated by the parsing algorithm of $O(n^3)$. Second, our approach is more general in the sense that any acyclic task graph can be analyzed. McCreary's method can handle only a few patterns of dependency relations among nodes. Third, our approach considers the execution and communication times as primary factors from the beginning and selects the best candidates from the entire task precedence graph based on the analysis of such times. As discussed before, McCreary's method, however, may lose the clustering opportunity at later stages because the decomposition of the graph is done without using such time information.

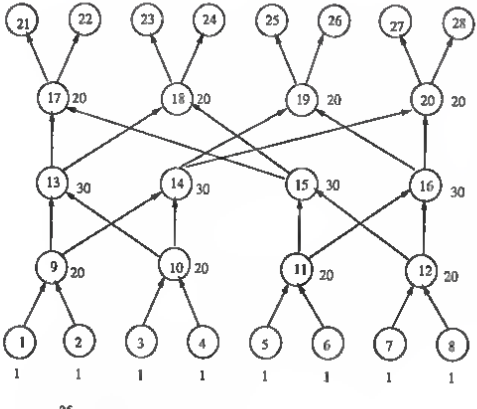


Figure 5.9 A task precedence graph for the FFT problem

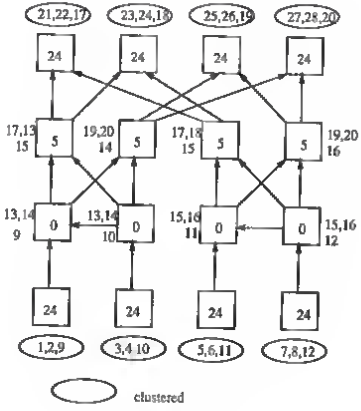


Figure 5.10 A gain graph for the FFT problem

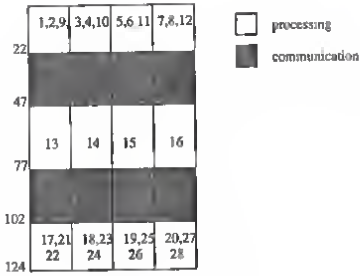


Figure 5 11 A Schedule for the FFT problem

CHAPTER 6
DISCUSSION

In this dissertation, we have presented (a) the computation model PROOF as a basis for expressing parallelism, (b) the IPR and the transformation rules as a basis for exploiting parallelism and (c) grain size determination techniques for three patterns of parallelism.

We have developed a computation model PROOF in which an object oriented paradigm is incorporated into a functional paradigm without sacrificing the advantages of either. PROOF allows the expression of parallelism on two different levels, object level and method level. The referential transparency and history sensitivity are integrated via the pseudo-function $\mathcal{R}$. One of the problems in a parallel object oriented computation model is to smoothly integrate synchronization constraints and inheritance. In PROOF, this integration is achieved by separating the synchronization constraint of each method from its pure operation.

We have developed an intermediate form IPR to make parallelism explicit in the PROOF/L programs. The IPR is a hybrid graphical representation in which object level behavior is represented as a Petri net and method-level behavior is represented as a set of function nodes and their data dependency relations. We have also developed transformation rules from the PROOF/L programs to the IPR and proved that the transformation preserves the correctness of the PROOF/L program by showing that the interleaving semantics of the PROOF/L program can be retrieved from the operational semantics of the IPR. The separation of semantics in the two different levels makes verification of the program easy. By introducing the IPR, the semantic

and performance issues of the program can be treated separately. The IPR can serve as a task precedence graph for grain size determination analysis. The programmer's responsibility to ensure correct synchronization and communication is reduced by inserting the proper code during the transformation phase. Considering the fact that coding of synchronization and communication is highly error-prone, this automatic coding reduce errors and thus benefit the software development.

We have presented a two-level allocation approach in which the objects are partitioned into a set of clusters and in each cluster the proper grain sizes are determined within each method. We have developed the grain size determination algorithms for different types of parallelism: pipe-lined, tree and graph. In the case of pipe-lined parallelism, we show that our algorithm can find optimal solutions. In the cases of tree-parallelism and graph-parallelism, we compare our approach to the existing methods and show that our approach can perform better in terms of the time complexity.

There are many potential directions for the future work. One immediate interest is to improve the performance of the target code by developing optimization techniques. We have experimented on various PROOF/L programs, including factorial problem, bounded buffer problem, dining philosopher problem, warehouse management problem and air base defense simulation problem. We have compared the performance of the Inmos C codes generated by our transformation system with the performance of the Inmos C codes written by hand in terms of the completion time. Because the current transformation system does not include optimization techniques, the performance of the generated code is not as good as the code implemented directly on the transputer systems. For example, we have extensively tested pipelined versions of the factorial program. The speed ups of the two factorial programs, the generated code and the directly written code by hand, are almost the same, but the absolute

completion time of the generated code is at best twice bigger than the completion time of the directly written code When the programs involve extensive manipulation of list data types, the generated codes require much more time than the written codes We believe that it is due to inefficiency of current list handling routines implemented in the back end transformation In addition to the code optimization techniques, we need to develop techniques that can reduce unnecessary data movement during object invocation. Another interesting aspect is to extend current work to real-time systems, such as process control systems, and to avionics In particular, its extension to distributed real-time application can be of immediate interest To do so, we will need to extend or modify the computation model in order to address real time specific issues, such as timing constraints and fault-tolerance In addition, scheduling and allocation strategies need to be changed, since in real time applications the goal of scheduling and allocation differs in that their goal is not to simply reduce the completion time but to schedule and allocate to meet the deadline of each task It will also be worthwhile to investigate how we can improve our computation model PROOF to allow dynamic creation of objects This extension will also require appropriate adjustments in the entire programming system, including adoptation of dynamic scheduling and allocation approaches We will also investigate the extension of the computation model with the array constructs so that arrays of objects can be easily created by programmers

Current transformation rules for the front-end translation are given based on lenient semantics, which is a very weak version of non strict semantics. In order to make our approach more versatile, we also need to develop different transformation rules for non strict semantics One way to detect parallelism in non strict semantics is to evaluate a function and its arguments at the same time Then the effect of the non-strict semantics can be determined In addition, the transformation rules

for exploiting conservative parallelism by lazy semantics need to be developed The comparison of these three ways of exploiting parallelism will also be of interest

Current implementation does not fully support the CREW model of execution, although the computation model PROOF supports such a model Each object is either considered as a read only object or a writable object, but objects can be combination of both. We will investigate a technique to fully support the CREW model, including efficient synchronization constraints for improving parallel execution

A1. Syntax for PROOF/L

```
proofl -> 'program' ID ' ' class_list obj_list body_list 'end'
body_list -> body_def body_list
   | body_def
body_def -> 'Body' 'of' 'object' ID '|' func
obj_list -> obj_list obj_def
  | obj_def
obj_def -> active_opt 'Object' id_list ' ' 'instance' 'of' ID
           '(' ins_list ')'
active_opt -> 'Active'
    |
ins_list -> ins_list ins
  | ins
ins -> INTEGER
     | data_type
class_list -> class_list class_def
     | class_def
class_def -> 'class' ID '(' dcls_list ')' var_list super_class
             method_def 'end' 'class'
super_class -> 'superclass' ' ' ID '(' dcls_list ')' 'inherit' '|'
               inherit_opt |
inherit_opt -> 'all'
     | id_list
id_list -> id_list ',' ID
 | ID
var_list -> composition var
   |
var -> var 'X' dcls
     | dcls
method_def -> method method_def
    | method
method -> 'method' ID '(' method_io ')' guard_dcls 'expression' func
method_io ->  input_list '->' output_list
   |
input_list -> dcls_list
dcls_list -> dcls ',' dcls_list
   | dcls
dcls -> ID ':' data_type
      | ID ':' ID
      | ID ':' list '(' data_type ')'
      | ID ':' list '(' ID ')'
      | ID
      | data_type
```

```
data_type -> 'int'
   | 'boolean'
output_list -> output ',' output_list
      | output
      |
guard_dcln -> 'guard' '(' bool_exp ')'
     |
func -> 'alpha' ID '[' func_list ']'
      | 'beta' '[' fuoc_list ']' '[' func_list ']'
      | 'gamma' '[' bool_exp_list ']' '[' func_list ']'
      | 'delta' '[' func_list ']' ID
      | 'sta'   ID '[' func_list ']'
      | stmt
stmt -> 'while' '(' bool_exp ',' func ')'
      | 'if' '(' bool_exp ',' func ',' func ')'
      | 'R' '[' ID ']' asaign_opt
      | exp
asaign_opt -> func
     |
exp -> aexp
     | binop aexp aexp
bool_exp ->  bool_op aexp aexp
   | 'not' bool_op
   | 'True'
   | 'False'
bool_exp_list -> bool_exp ',' bool_exp_list
       | bool_exp
aexp -> ID
      | INTEGER
      | STRING
      | '(' func_list ')'
      | '[' func_list ']'
func_list -> func ',' func_list
   | func
binop -> '+'
       | '-'
       | '*'
       | '/'
       | 'and'
bool_op -> '='
 | '<>'
 | '<'
 | '<='
 | '>'
 | '>='
```

A2 PROOF/L program examples, their IPRs and generated target codes[1]

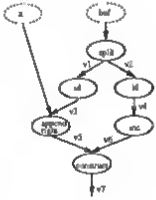
Bounded Buffer PROOF/L program[2]

```
Program Bounded_Buffer
class buffer (itemtype, size int)
    composition
        store list(itemtype) X count int
    method get (buf buffer → buffer, itemtype)
        guard (> buf count 0)
        expression
            [#[tail, dec][buf.store, buf count], head(buf.store)]
    method put (buf buffer, x itemtype → buffer)
        guard (< buf count size)
        expression
            β[append_right(x), inc][buf store, buf count]
    method is_empty (buf buffer → boolean)
        expression
            = buf count 0
    method length (buf buffer → int)
        expression
            buf count
end class
class producer (itemtype)
    method produce ( → itemtype]
        ...
end class
class consumer (itemtype)
    method consume (itemtype → )
        ...
end class
Object buf  instance of buffer
Active Object producer  instance of prodocer
Active Object consumer  instance of consumer
Body of object producer
    while (True, R [ buf ]  put(buf, produce()))
Body of object consumer
    while (True, β [R [ buf ]  consume()][get(buf)])
end
```

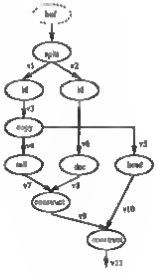
[1]The target code only for the bounded buffer problem is included

[2]This program is different from the one shown in Chapter 3 since the class interface is embedded into the class definition

An IPR for put



An IPR for get



A generated Iomos C code corresponding to a cluster of *producer* and *buffer*

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <channel.h>
#include <process.h>
# include "class.h"
# include "table0.h"
# include "router.h"
# include "vis.h"
# include "hb1.c"

int block_buf = FALSE,
int block_buf = FALSE,

void _buf(Process *P, Channel *chan01, Channel *synch,
                      Channel *Fromproc,Channel *Toproc[])
{
   int signal,
    MESG *message,
   int size = 1,
   int i =0,
   int j=0,
   int k=0,
   int var = 2; /* for router */
   P = P,
   buf count = 0,
   synch = synch,
    message = (MESG *)malloc (sizeof(MESG));
   for(;;){
   if (buf count < 1)
   {
     signal = ProcAlt(chan01,NULL),
     Print(("Hi\n"));
   }
   else if (buf count == size)
   {
     signal = ProcAlt(Fromproc[0],NULL),
     signal = 1,
   }
   else
     signal = ProcAlt(Fromproc[0],chan01, NULL), /* to be modified */
   switch(signal) {
      case 0 /* channel chan01 ready */
      {
         i++,
         Print(("Producer %d\n",i)),
         ChanInlet(chan01),
         ChanOut(chan01, buf store, MSIZE * 4),
         ChanOutlet(chan01, buf count),
```

```
        ChanIn(chan01, buf store, NSIZE * 4),
        buf count = ChanInInt(chan01),
        break,
      }

      case 1  /* file 3 ready */
      {
        j++,
        Print(("Consumer %d\n",j)),
        receive(Fromproc[0],message,&var),
        make_message(message,1,0,200),
        send(Toproc[0],message,buf store),
        make_message(message,1,0,4),
        send(Toproc[0],message,&buf count),
        receive(Fromproc[0],message,buf store),
        receive(Fromproc[0],message,&buf_count),
        break,
      }

   }
   Print(("\nbuf_count = %d\n",buf_count)),
   Print(("Buffer\n")),
   fflush(stdout),
   for (k=0,k<buf_count,k++)
    Print((" %d ",buf store[k])),
   Print(("\n***************\n"));
}

}


void _producer(Process *p, Channel *chan01, Channel *synch,Channel
*Fromproc[])
{
   /*struct buffer lv002,*/
   struct buffer lv003,
   MESG *message,
   int lv004,
   int i,
   int var = 2, /* for router */

   p = P,
   message = (MESG *)malloc(sizeof(MESG)),
   synch = synch,
   for(i=0,i<20,i++)
   {
     produce(&producer, &lv004),
     ChanOutInt(chan01,var),
     ChanIn(chan01, buf store, NSIZE * 4),
     buf count = ChanInInt(chanOI),
     put(&buf, lv004, &lv003),/* Changed the 3rd parametar */
```

```
    ChanOut(chan01, lv003 store, NSIZE * 4),
    ChanOutInt(chan01,lv003 count),
  }
  receive(Fromproc[1],message,&i);
  fflush(stdout),
  exit_terminate(EXIT_SUCCESS),
}


void _main(Process *p,Channel *Fromproc[],Channel *Toproc[])
{
  Process *p01, /* buf */
  Process *p02, /* producer */
  Channel *chan01, /* producer <-> buf */
  Channel *synch, /* Synchronization channel */

  p = p,
  chan01 = ChanAlloc(),
  if (chan01 == NULL)
    abort(),
  synch = ChanAlloc(),
  if (synch == NULL)
    abort(),
  p01 = ProcAlloc(_buf, 0, 4, chan01, synch,Fromproc,Toproc),
  if (p01 == NULL)
    abort(),
  p02 = ProcAlloc(_producer, 0, 3, chan01, synch,Fromproc),
  if (p02 == NULL)
    abort(),

  buf store = nil();

  ProcPar(p01, p02, NULL),

}


int main()
{
  Process *control,
  Process *tt,
  Channel *Toproc[SOFT_CHANNELS+1],
  Channel *Fromproc[SOFT_CHANNELS+1],
  int i,
  set_abort_action(ABORT_HALT),
  for (i=0,i< SOFT_CHANNELS,i++)
  {
    if ((Fromproc[i] = ChanAlloc()) == NULL)
      abort(),
  }
  for (i=0,i< SOFT_CHANNELS,i++)
```

```
  {
    if ((Toproc[i] = ChanAlloc()) == NULL)
      abort();
  }
  Fromproc[SOFT_CHANNELS] = NULL;
  Toproc[SOFT_CHANNELS] = NULL;
  control = ProcAlloc(Router,0,2,Fromproc,Toproc);
  if (control == NULL)
    abort();
  tt = ProcAlloc(_main,0,2,Fromproc,Toproc);
  if (tt == NULL)
  {
    abort();
  }
  /*init_trace(3);*/
  ProcPar(control,tt,NULL);
}
```

A generated Inmos C code corresponding to a cluster of *consumer*

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <channel.h>
#include <process.h>
# include "class.h"
# include "table1.h"
# include "routered.h"
# include "vis.h"
# include "fhi.c"

void _consumer(Process *p, Channel *synch,Channel *Fromproc[],Channel *Toproc
{
   Channel *synch2;
   struct buffer lv003;
   struct buffer lv004;
   struct buffer vbuf;
    MESG * message;
   int lv005;
   int var; /* for router */
   int i;
   p = p;

   synch2 = ChanAlloc();
   if (synch2 == NULL)
      abort();
    message = (MESG *)malloc(sizeof(MESG));
   var = 10101;
   vbuf.store = nil();
   for (i=0;i<20;i++)
   {
     make_message(message,0,0,4);
     send(Toproc[0],message,&var);
     receive(Fromproc[0],message,vbuf.store);
     receive(Fromproc[0],message,&vbuf.count);
     get(&vbuf, &lv004,&lv005);
     consume(&consumer,lv005);
     make_message(message,0,0,200);
     send(Toproc[0],message,lv004.store);
     make_message(message,0,0,4);
     send(Toproc[0],message,&lv004.count);

   }

   ChanOutInt(synch, 1);
}


void _main(Process *p,Channel *Fromproc[],Channel *Toproc[])
```

```
{
   Process *p01, /* consumer */
   Channel *synch; /* Synchronization channel */

   synch = ChanAlloc(),
   if (synch == NULL)
      abort(),

   p01 = ProcAlloc(_consumer, 0, 3, synch,Fromproc,Toproc),
   if (p01 == NULL)
      abort(),

   ProcRun(p01),

   ChanInInt(synch),
}

int main()
{
  Process *control,
  Process *tt,
  Channel *Toproc[SOFT_CHANNELS+1],
  Channel *Fromproc[SOFT_CHANNELS+1],
  int i,
  set_abort_action(ABORT_HALT),
  for (i=0,i< SOFT_CHANNELS,i++)
  {
    if ((Fromproc[i] = ChanAlloc()) == NULL)
      abort(),
  }
  for (i=0,i< SOFT_CHANNELS,i++)
  {
    if ((Toproc[i] = ChanAlloc()) == NULL)
      abort(),
  }
  Fromproc[SOFT_CHANNELS] = NULL,
  Toproc[SOFT_CHANNELS] = NULL,
  control = ProcAlloc(Router,0,2,Fromproc,Toproc),
  if (control == NULL)
  abort(),
  tt = ProcAlloc(_main,0,2,Fromproc,Toproc),
  if (tt == NULL)
  {
    abort(),
  }
  ProcPar(control,tt,NULL),
}
```

A generated Inmos C code corresponding to the method definitions

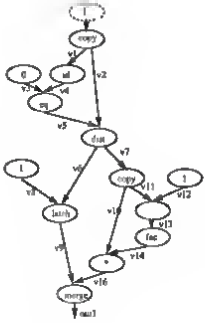
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <channel.h>
#include <process.h>
#include "primitive.h"
#include <misc.h>

void get(struct buffer *buf, struct buffer *out01, int *out02)
{
   int lv006,
   int lv010,
   int *lv002,
   dec(buf->count, &lv006),
   head(buf->store, &lv010),
   lv002 = nil(),
   tail(buf->store, lv002),
   out01 -> store = lv002,
   out01 -> count = lv006,
   *out02 = lv010,

}


void put(struct buffer *buf, int x, struct buffer *out01)
{
   int *lv001,
   int *lv002,
   int lv003,
   int lv004,
   int *lv005,
   int lv006,

   lv005 = buf->store,
   lv006 = buf->count;
   lv004 = lv006,
   inc(lv004, &lv003),
   lv002 = lv005,
   lv001 = nil(),
   append_right(x, lv002, lv001);
   out01->store = lv001,
   out01->count = lv003,

}


void produce(struct producer_class *producer, int *out01)
{
```

```
    *out01 = 10,
    producer = producer,
    /*produce(&producer, out01),*/

}


void consume(struct consumer_class *consumer, int x)
{
    consumer = consumer,
    x = 0,
    /*consume(&consumer, x);*/

}
```

Factorial PROOF/L program

```
program fact

class t1(int)
      method fac ( x int -> int)
      expression
            if((lambda (x) = 0 x),1,(lambda (x) * x (fac (- x 1))))x
end class

Active Object tt   instance of t1

Body of object tt | fac 0

end
```

An IPR for fac



Dining Philosopher PROOF/L program

```
program dining_philosopher


clase chopstick

composition
        stick   int

method acquire(c chopstick -> chopstick)
        guard ( = c stick 0)
        expression
                inc stick

method acquire(c chopstick -> chopstick)
        guard ( = c stick 1)
        expression
                dec stick

end clase


clase philosopher

composition
        dummy : int

method think( p philosopher -> int )
        expression
                delay(3)

method eat(  p philosopher -> int )
        expression
                delay(3)

end clase

Active Object p1  : instance of philosopher
Active Object p2  : instance of philosopher
Active Object p3  : instance of philosopher
Active Object p4  : instance of philosopher
Active Object p5  : instance of philosopher

Object c1  : instance of chopstick
Object c2  : instance of chopstick
Object c3  : instance of chopstick
Object c4  : instance of chopstick
Object c5  : instance of chopstick
```

```
Body of object p1

while(True,,(R[|c2|] c2 acquire(c2), R[|c1|] c1 acquire(c1), eat(p1),
             R[|c2|] c2 release(c2), R[|c1|]
c1 release(c1),think(p1)))  0


Body of object p2

while(True,,(R[|c2|] c2 acquire(c2), R[|c3|] c3 acquire(c3), eat(p2),
             R[|c3|] c2 release(c3), R[|c2|]
c2 release(c2),think(p2)))  0


Body of object p3

while(True,,(R[|c3|] c3 acquire(c3), R[|c4|] c4 acquire(c4), eat(p3),
             R[|c3|] c3 release(c3), R[|c4|]
c4 release(c4),think(p3)))  0


Body of object p4

while(True,,(R[|c4|] c4 acquire(c4), R[|c5|] c5 acquire(c5), eat(p4),
             R[|c4|] c4 release(c4), R[|c5|]
c5 release(c5),think(p4)))  0


Body of object p5

while(True,,(R[|c5|] c5 acquire(c5), R[|c1|] c1 acquire(c1), eat(p1),
             R[|c1|] c1 release(c1), R[|c5|]
c5 release(c5),think(p5)))  0
end
```

An IPR for Dining Philosopher
In the following we present the textual representation of an active object p1

Active Object p1 : philosopher

File 1 macro

Node Number	Node Name	Number of Input	Input Node	Number of Output	Output Node Number
55	DIST	2	29,54	2	31,OUTPUT
29	TRUE	1	54	1	55
54	COPY	1	53	2	29,55
53	MERGE	2	44,52	1	54
52	0	0	INPUT	1	53
44	think	1	45	1	53
45	LATCH	2	43,p1	1	44
43	ASSIGN	1	41	2	c1,45
41	release	1	42	1	43
42	LATCH	2	40,c1	1	41
40	ASSIGN	1	38	2	c2,42
38	release	1	39	1	40
39	LATCH	2	36,c2	1	38
36	eat	1	37	1	39
37	LATCH	2	35,p1	1	36
35	ASSIGN	1	33	2	c1,37
33	acquire	1	34	1	35
34	LATCH	2	32,c1	1	33
32	ASSIGN	1	30	2	c2,34
30	acquire	1	31	1	32
31	LATCH	2	55,c2	1	30

;

REFERENCES

[1] T L Adam, K M Chandy, and J R Dickson A comparison of list schedules for parallel processing systems *Communications of the ACM*, 17(12) 685–690, 1974

[2] G Agha. *Actors: A Model of Concurrent Computation in Distributed Systems* MIT Press, Cambridge,Massachusetts, 1986

[3] F Allen, M Burke, P Charles, R. Cytron, and J Ferrante An overview of the PTRAN analysis system for multiprocessing *Journal of Parallel and Distributed Computing*, 5(5) 617–640, October 1988

[4] P America A parallel object-oriented language In A Yonezawa and M Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220 MIT Press, 1987

[5] Ardent Computer Co *Programmer's Guide*, 1989

[6] Z Ariola and Arvind P-TAC A parallel intermediate language In *Proc of the Functional Programming Languages and Computer Architecture*, pages 230–242, 1989

[7] Arvind and K Ekanadham Future scientific programming on parallel machines *Journal of Parallel and Distributed Computing*, 5(5) 460–493, Oct 1988

[8] J Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs *Communications of the ACM*, 21(8) 613–641, Aug 1978

[9] H E. Bal, Steiner J G , and A S Tanenbaum Programming languages for distributed computing systems *ACM Computing Surveys*, 21(3) 261–322, 1989

[10] H P Barendregt, M C J D. van Eekelen, J R W Glauert, M J. Plasmeijer, and M R. Sleep Towards an intermediate language based on graph rewriting In *Proc of the PARLE Conference, LNCS 259*, pages 159–175, 1987

[11] J Beetam, M Denneau, and D Weingarten The GF11 supercomuter In *Proc of the 12th Annual International Symposium on Computer Architecture*, pages 108–115, 1985

[12] A Black, N Hutchinson, E. Jul, and H Levy Object structure in the Emerald system In *Proc of Object Oriented Programming Systems, Languages and Applications. ACM SIGPLAN Notices 21(11)*, pages 78–86, 1986.

[13] P Brinch Hansen The programming language concurrent Pascal *IEEE Transactions on Software Engineering*, 1(2) 199–207, June 1975

[14] L Brownston *Programming Expert Systems in OPS5* Addison-Wesley, Reading, Massachusetts 1985

[15] N Carriero and D Gelernter Linda in context *Communications of the ACM*, 32(4) 444–458, 1989

[16] C Y Chin and K Hwang Packet switching networks for multiprocessors and dataflow computers *IEEE Trans on Computer*, C-33(11)-991–1003, 1984

[17] W W Chu, L J Holloway, M T-Lan and K Efe Task allocation in distributed data processing *IEEE Computer*, 13(11) 57–69, 1980

[18] A Church and J B Rosser Some properties of conversion *Transactions of American Mathematics Society*, 39 472–482, 1936

[19] K L Clark and S Gregory PARLOG Parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1) 1–49, 1986

[20] Cray Research Inc *Fortran (CFT) Reference Manual*, 1984

[21] A L Davis and R M Keller Data flow program graphs *Computer*, 15(2) 26–41, 1982

[22] Department of Defence *Reference Manual for the Ada Programming Language*, 1983 ANSI/MIL-STD-1815A-1983 1983

[23] E W Dijkstra Guarded Commands, nondeterminacy and formal derivation of programs *Communications of the ACM*, 18 453–457, 1975

[24] K Efe. Heuristic models of task assignment scheduling in distributed systems *IEEE Computer*, 15(6) 50–56, 1982.

[25] O I El Dessouki and W H Huen Distributed enumeration on network computers *IEEE Transactions on Computers*, C 29(8) 818–825, 1980

[26] K P Eswaran, J N Gray, R A Lorie, and I L Traiger The notions of consistency and predicate locks in a database system *Communications of the ACM* 19(11) 624–633, Nov 1976

[27] S Fahlman *NETL A System for Representing and Using Real World Knowledge* MIT Press, Cambridge, Massachusetts, 1979

[28] P R Fenner The Flex/32 for real time multicomputer simulation In W J Karplus, editor, *Multiprocessors and Array Processors*, pages 127–136. Simulation Councils, Inc , 1987

[29] M J Flynn Some computer organizations and their effectiveness *IEEE Transactions on Computers*, C-21(9) 948–960, 1972

[30] M J Fortune Parallelism in random access machines In *Proc 10th ACM STOC*, pages 114–118, 1978

[31] I Foster. *Strand: New Concepts in Parallel Programming*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[32] M R Garey and D S Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, San Francisco, CA, 1979.

[33] J-L Gaudiot and L-T Lee. Occamflow: A methodology for programming multiprocessor systems. *Journal of Parallel and Distributed Computing*, 7:96–124, 1989.

[34] B Goldberg. Multiprocessor execution of functional programs. *Journal of Parallel Programming*, 17(5):425–473, 1988.

[35] J Graham and J Rattner. Expert computation on the iPSC concurrent computer. In W. J. Karplus, editor, *Multiprocessors and Array processors*, pages 167–178. Simulation Councils, Inc., 1987.

[36] A S Grimshaw and J W S Liu. Mentat: An object oriented macro data flow system. In *Proc. of Object Oriented Programming Systems, Languages and Applications 1987. ACM SIGPLAN Notices 22(12)*, pages 35–47, 1987.

[37] V B Gylys and J A. Edwards. Optimal partitioning of workload for distributed systems. In *Digest of Papers, COMPCON Fall 76*, pages 353–357, 1976.

[38] R. H. Jr Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming languages and Applications*, 7(4):501–538, 1985.

[39] W. D. Hillis. *The Connection Machine*. The MIT Press, Cambridge, Massachusetts, 1985.

[40] C A R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug 1978.

[41] R. Hockney and C. Jesshope. *Parallel Computers*. Adam Hilger, 1988.

[42] P. Hudak. Para-functional programming. *IEEE Computer*, 19(8):60–70, 1986.

[43] C. Huson, T. Mache, J. Davies, M. Wolfe, and B. Leasure. The KAP/205: An advanced source-to-source vectorizer for the Cyber 205 supercomputer. In *Proc. of International Conference on Parallel Processing*, pages 827–832, 1986.

[44] F. J. Hwang, F. D. Anger, Y. C. Chow, and C. Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, 1989.

[45] K. Hwang. Advanced parallel processing with supercomputer architecture. *Proceedings of the IEEE*, 75(10):1348–1379, 1987.

[46] Inmos Ltd. *Occam Programming Manual*, 1984.

[47] R. Jagannathan, A. R. Downing, W. T. Zaumen, and R. K. S. Lee. Dataflow-based methodology for coarse-grain multiprocessing on a network of workstations. In *International Conference on Parallel Processing*, volume II, pages 209–209, 1989.

[48] R. Jagannathan and A. A. Faustini. The GLU programming language. Technical Report SRI CSL-90-11, SRI International Computer Science Laboratory, 1990.

[49] D. C. Kafura and K. H. Lee. Inheritance in actor based concurrent object-oriented languages. In *Proc. of the 3rd European Conference on Object-Oriented Programming*, pages 131–145, 1989.

[50] H. Kasahara and N. Seinosuke. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, C-33(11):1023–1029, 1984.

[51] R. M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19:371–384, 1976.

[52] S. J. Kim. A general approach to multiprocessor scheduling. Technical report, University of Texas at Austin, 1988.

[53] H. T. Kung. Notes on VLSI computation. In D. J. Evans, editor, *Parallel Processing Systems*. Cambridge University Press, 1982.

[54] H. Liberman. Concurrent object oriented programming in Act 1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press, 1987.

[55] V. M. Lo. Heuristic algorithms for task assignment in distributed systems. In *Proc. 4th Int'l Conf. on Distributed Computing Systems*, pages 30–39, 1984.

[56] Inmos Ltd. *The Transputer Databook*. Inmos Ltd., Bristol, UK, 1989.

[57] P. R. Ma and E. Y. S. Lee. A task allocation model for distributed computing systems. *IEEE Transactions on Computers*, C-31(1):41–47, 1982.

[58] C. McCreary and H. Gill. Automatic determination of grain size for efficient parallel processing. *Communications of the ACM*, 32(9):1073–1078, 1989.

[59] C. McCreary and H. Gill. Efficient exploitation of concurrency using graph decomposition. In *International Conference on Parallel Processing*, pages 199–203, 1990.

[60] J. McGraw. SISAL: streams and iteration in a single assignment language, language reference manual, version 1.2. Technical Report Technical Report M-146, LLNL, 1985.

[61] O. M. Nierstrasz. Active objects in hybrid. In *Proc. of Object-Oriented Programming Systems, Languages and Applications*, pages 243–253, 1987.

[62] R. S. Nikhil, K. Pingali, and Arvind. Id Nouveau. Technical Report Computation Structures Group Memo 265, Laboratory for Computer Science, MIT, 1986.

[63] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of ACM*, 26(4):631–653, Oct 1979.

[64] J F Palmer The NCUBE family of parallel supercomputers In W J Karplus, editor, *Multiprocessors and Array processors*. Simulation Councils, Inc, 1987

[65] J L Peterson *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981

[66] G F Pfister, W C Brantley, D A George, S L Harvey, K P McAuliffe, and E A Melton The ibm research parallel processor prototype (rp3) In *Proc of International Conference on Parallel Processing*, pages 764–771, 1985

[67] C D Polychronopoulos and U Banerjee Processor allocation for horizontal and vertical parallelism and related speedup bounds *IEEE Transactions on Computers*, C-36(4) 410–420, 1987

[68] C C Price and S Krishnaprasad Software allocation models for distributed computing systems In *International Conference on Distributed Computing Systems*, pages 40–48, 1984

[69] C V Ramamoorthy, J Srivastava, and W T Tsai Clustering techniques for large distributed systems In *IEEE INFOCON'86*, pages 395–404, 1986

[70] V Sarkar and V Hennessy Partitioning parallel programs for macro-dataflow In *ACM Conference in Lisp and Functional Programming*, pages 202–211, August 1986

[71] E Shapiro Concurrent Prolog A progress report *IEEE Computer*, 19(8) 44 58, 1986

[72] C C Shen and W T Tsai A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion *IEEE Transactions on Computers*, 34(3) 197–203, 1985

[73] S K Skedzielewski and M L Welcome Data flow graph optimization in IF1 In *Proc of Functional Programming Languages and Computer Architecture Conference*, pages 17–34, 1985

[74] H S Stone Multiprocessor scheduling with the aid of network flow algorithms *IEEE Transactions on Software Engineering*, SE-3(1) 85–93, 1977

[75] H S Stone and S H Bokhari Control of distributed processes *IEEE Computer*, 11(7) 97–106, 1978

[76] J Test, M Myszewski, and R C Swift The Alliant FX/Series Automatic parallelism in a multiprocessor mini supercomputer In W J Karplus, editor, *Multiprocessors and Array processors*, pages 35–44. Simulation Councils, Inc, 1987

[77] S Thakkar, P Gifford, and G Fielland Balance A shared memory multiprocessor In *Proc of the Second International Conference on Supercomputing*, pages 23–30, 1987

[78] C Tomlinson and V Singh Inheritance and synchronization with enabled-sets. In *Proc of Object Oriented Programming Systems, Languages and Applications*, pages 103–112, 1989

[79] K R. Traub. Compilation as partitioning: A new approach to compiling non-strict functional languages. In *Proc. of the Functional Programming Languages and Computer Architectures*, pages 75–88, September 1989.

[80] J. L. Uhrig. Mathematical programming approaches to system partitioning. *IEEE Transactions on Systems, Man, Cybernetics*, SMC-8:540–548, 1978.

[81] J. D. Ullman. NP-complete scheduling problems. *Journal of Comput. Syst. Sci.*, 10:384–393, 1975.

[82] S. S. Yau, D. H. Bae, and M. Chidambaram. A framework for software development for distributed parallel computing systems. In *Proc. of the third Workshop on the Future Trends of Distributed Computing Systems*, April 1992.

[83] S. S. Yau, X. Jia, and D.-H. Bae. Trends in software design for distributed computing systems. In *Proc. of the Second Workshop on the Future Trends of Distributed Computing Systems*, pages 154–160, 1990.

[84] S. S. Yau, X. Jia, and D.-H. Bae. PROOF: A parallel object-oriented functional computation model. *Journal of Parallel and Distributed Computing*, 12(3):202–212, July 1991.

[85] S. S. Yau, X. Jia, and D.-H. Bae. Software design methods for distributed computing systems. *Journal of Computer Communications*, 1992.

[86] S. S. Yau, X. Jia, D. H. Bae, M. Chidambaram, and G. Oh. An object-oriented approach to software development for parallel processing systems. In *Proc. International Computer Software and Applications Conference (COMPSAC'91)*, pages 453–458, 1991.

[87] S. S. Yau and C.-S. Liu. A structured inheritance network representation for object-oriented software design. In *Proc. of 13th Annual Int'l Computer Software & Applications Conference (COMPSAC'89)*, pages 351–357, 1989.

[88] Y. Yokote and M. Tokoro. Concurrent programming in concurrent Smalltalk. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 129–158. MIT Press, 1987.

[89] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Proc. of Object-Oriented Programming Systems, Languages and Applications: ACM SIGPLAN Notices 21(11)*, pages 258–268, 1986.

BIOGRAPHICAL SKETCH

Doo-Hwan Bae was born on December 5, 1957, in Seoul, Korea. He received the B.S. and M.S. degrees in ocean engineering from the Seoul National University, Seoul, Korea, in 1980 and 1982, respectively. He completed his military duty in May 1983, and worked as a Researcher at Korea Advanced Institute of Science and Technology until May 1984. He received the M.S. degree in computer science from the University of Wisconsin-Milwaukee in 1987. He studied in the Computer Science Program at the Northwestern University from September 1987 to August 1988. He is currently a Ph.D. candidate in the Computer and Information Sciences Department, University of Florida, Gainesville.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy

Stephen S. Yau, Chair
Professor of Computer and
Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy

Yau Hang Lee
Associate Professor of Computer
and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy

Paul W. Chun
Professor of Biochemistry
and Molecular Microbiology

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Paul A Fishwick

Paul A. Fishwick
Associate Professor of Computer
and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy

Tim Davis

Tim Davis
Assistant Professor of Computer
and Information Sciences

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

May 1992

Winfred M. Phillips
Dean, College of Engineering

Madelyn M. Lockhart
Dean, Graduate School